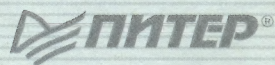
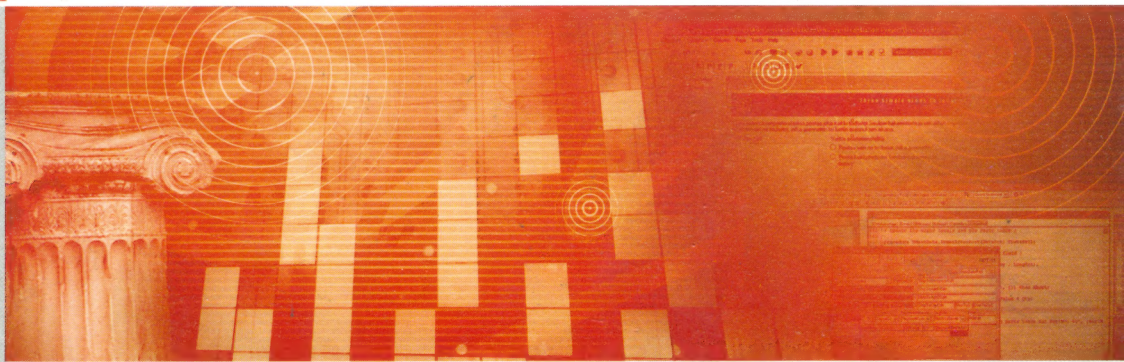


УЧЕБНИК
ДЛЯ ВУЗОВ

 ПИТЕР®

В. В. Фаронов



Delphi

Программирование
на языке высокого
уровня

ДОПУЩЕНО
МИНИСТЕРСТВОМ ОБРАЗОВАНИЯ И НАУКИ РФ

С Е Р И Я

УЧЕБНИК // ДЛЯ ВУЗОВ





В. В. Фаронов

Delphi

Программирование на языке высокого уровня

Допущено Министерством образования Российской Федерации
в качестве учебника для студентов высших учебных заведений,
обучающихся по направлению подготовки дипломированных специалистов
«Информатика и вычислительная техника»



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2010

ББК 32.973-018.2

УДК 681.3.068

Ф24

Рецензенты:

Еркович С. П., доктор физико-математических наук,
профессор Московского государственного технического университета им. Н. Э. Баумана

Павловская Т. А., кандидат технических наук, доцент кафедры информатики и прикладной математики
Санкт-Петербургского института точной механики и оптики (технического университета)

Фаронов В. В.

Ф24 Delphi. Программирование на языке высокого уровня: Учебник для вузов — СПб.: Лидер, 2010. — 640 с.: ил.

ISBN 978-5-8046-0008-3

Книга посвящена новейшей версии Delphi 7 Studio. Здесь изложены как приемы программирования в среде Delphi, ее главные составные части — галереи компонентов, хранилища объектов, вспомогательный инструментарий, так и сам язык программирования Delphi. Подробно рассматриваются компоненты программ, некоторые дополнительные возможности — динамически подключаемые библиотеки, интерфейсы, технология COM и система ModelMaker.

Книга может быть полезна как начинающим — в качестве пособия для первоначального изучения среды и языка Delphi, так и опытным программистам, желающим пополнить свои знания в области применения языка Delphi.

Допущено Министерством образования Российской Федерации в качестве учебника для студентов высших учебных заведений, обучающихся по направлению подготовки дипломированных специалистов «Информатика и вычислительная техника».

ББК 32.973-018.2

УДК 681.3.068

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-8046-0008-3

© ООО «Лидер», 2010

Краткое содержание

От автора	17
Введение	18
Часть I. ЗНАКОМСТВО С DELPHI	24
Глава 1. Среда разработчика	25
Глава 2. Основы визуального программирования	38
Глава 3. Знакомство с компонентами	48
Глава 4. Введение в язык Delphi	55
Часть II. ЯЗЫК DELPHI	102
Глава 5. Элементы языка	103
Глава 6. Типы данных	110
Глава 7. Процедуры и функции	152
Глава 8. Классы	170
Глава 9. Варианты	184
Глава 10. Файлы	192
Глава 11. Модули	214
Глава 12. Типизированные константы и инициализированные переменные	222
Часть III. КОМПОНЕНТЫ И ПРОГРАММА	228
Глава 13. Классы общего назначения	229
Глава 14. Общие свойства компонентов	261

Глава 15. Компоненты вкладки Standard	293
Глава 16. Компоненты вкладки Additional	317
Глава 17. Компоненты вкладки Win32	353
Глава 18. Компоненты вкладки System	400
Глава 19. Компоненты вкладки Dialogs	419
Глава 20. Компоненты вкладки Samples	432
Глава 21. Форма	439
Глава 22. Программа	454
 Часть IV. РАСШИРЕННЫЕ ВОЗМОЖНОСТИ DELPHI	 502
Глава 23. Создание и использование DLL	503
Глава 24. Интерфейсы	516
Глава 25. Технология COM	525
Глава 26. Создание компонентов с помощью ModelMaker	542
Приложение А. Интегрированная среда разработчика	568
Приложение Б. Некоторые стандартные подпрограммы, переменные, классы	607
Литература	628
Алфавитный указатель	629

Содержание

От автора	17
Введение	18
Назначение и возможности Delphi	18
Инструмент ускоренной разработки программ	18
Мощность и гибкость языка	20
Инструмент создания приложений баз данных	21
Особенности версии Delphi 7 Studio	21
От издательства	23
Часть I. ЗНАКОМСТВО С DELPHI	24
Глава 1. Среда разработчика	25
Главное окно	27
Инструментальные кнопки	28
Настройка окон	30
Палитра компонентов	30
Окно формы	31
Окно дерева объектов	32
Окно инспектора объектов	32
Окно кода программы	34
Глава 2. Основы визуального программирования	38
Пустая форма и ее модификация	38
Настройка Delphi	38
Имена в Delphi	40
Изменение свойств формы	40
Размещение нового компонента	41
Реакция на события	43
Модификация формы	43

Обработчик события OnClick	43
Динамическое изменение свойств компонента	46
Некоторые итоги	47
Глава 3. Знакомство с компонентами	48
Вкладка Standard	48
Вкладка Additional	49
Вкладка Win32	51
Вкладка System	52
Вкладка Dialogs	53
Вкладка Win31	53
Вкладка Samples	54
Вкладка ActiveX	54
Глава 4. Введение в язык Delphi	55
Учебная программа	55
Конструирование формы	56
Перенос формы в хранилище	59
Структура программ Delphi	61
Структура проекта	61
Структура модуля	64
Элементы программы	66
Типы	69
Строковый и символьный типы	69
Целые типы	73
Операторы языка	77
Составной оператор и пустой оператор	77
Условный оператор	78
Операторы повторений	82
Оператор выбора	86
Метки и операторы перехода	90
Массивы	91
Процедуры и функции	94
Правила кодирования программ	98
 Часть II. ЯЗЫК DELPHI	 102
Глава 5. Элементы языка	103
Алфавит	103
Идентификаторы	104
Константы	105
Выражения	106
Операции	107

Глава 6. Типы данных	110
Простые типы	111
Порядковые типы	111
Вещественные типы	119
Тип дата-время	123
Структурированные типы	124
Массивы	125
Записи	129
Множества	132
Строки	136
Указатели и динамическая память	146
Динамическая память	146
Указатели	146
Выделение и освобождение динамической памяти	147
Псевдонимы типов	150
Типы с управляемым временем жизни	151
Глава 7. Процедуры и функции	152
Локализация имен	152
Описание подпрограммы	156
Заголовок и стандартные директивы	156
Параметры	157
Умалчиваемые параметры	162
Параметры-массивы и параметры-строки	162
Открытые массивы	163
Конструктор массива	164
Вариантные массивы-параметры	165
Процедурные типы	166
Рекурсия и опережающее описание	167
Глава 8. Классы	170
Основные понятия	171
Инкапсуляция	171
Наследование	171
Полиморфизм	172
Составляющие класса	172
Поля	172
Методы	173
Одноименные методы	177
Свойства	179
Объявление класса	180
Приведение типов классов	183

Глава 9. Варианты	184
Преобразование вариантов к данным других типов	186
Использование вариантов в выражениях	186
Подпрограммы для работы с вариантами	187
Вариантные массивы	188
Пользовательские варианты	190
Глава 10. Файлы	192
Доступ к файлам	193
Подпрограммы для работы с файлами	194
Текстовые файлы	199
Типизированные файлы	203
Нетипизированные файлы	203
Отображение файлов в память	205
Объектная модель работы с файлами	211
Глава 11. Модули	214
Структура модулей	215
Заголовок модуля и связь модулей друг с другом	215
Интерфейсная часть	216
Исполняемая часть	216
Инициализирующая и завершающая части	217
Доступ к объявленным в модуле объектам	218
Типы модулей в Delphi	220
Модули данных	221
Модули динамических библиотек	221
Пакеты	221
Модули потоков команд	221
Глава 12. Типизированные константы и инициализированные переменные	222
Типизированные константы	222
Константы простых типов и типа String	223
Константы-массивы	224
Константы-записи	225
Константы-множества	226
Константы-указатели	226
Инициализация переменных	226
Часть III. КОМПОНЕНТЫ И ПРОГРАММА	228
Глава 13. Классы общего назначения	229
Класс Exception — обработка исключений	229
Защищенные блоки	229

Класс Exception	231
Стандартные классы исключений	232
Вызов исключения	233
Создание собственного класса	233
Класс TList — списки	235
Классы TStringList и TStringList — наборы строк и объектов	239
Класс TStringList	239
Класс TStringList	240
Графический инструментарий	243
Класс TFont	243
Класс TPen	246
Класс TBrush	250
Класс TCanvas	253
Классы TGraphic и TPicture	257
Глава 14. Общие свойства компонентов	261
Иерархия компонентов	261
Имена и владельцы компонентов	262
Родительские и дочерние компоненты	264
Положение, размеры и оформление компонентов	267
Указатели мыши	274
Реакция на события мыши и клавиатуры	278
События мыши	278
События клавиатуры	279
Клавиатура в Windows	280
Фокус ввода	282
Механизм действий	283
Механизм перетаскивания	283
Механизм причаливания	286
Поддержка справочной службы	288
Поддержка технологии COM	290
Свойства разного назначения	291
Глава 15. Компоненты вкладки Standard	293
TFrame — фрейм и шаблоны компонентов	293
TMainMenu — главное меню формы	296
TPopupMenu — контекстное меню	298
TLabel — метка	298
TEdit — однострочное редактируемое текстовое поле	299
TMemo — многострочное редактируемое текстовое поле	302
TButton — кнопка	304
TCheckBox — флажок	305

TRadioButton — переключатель	306
TListBox — список	306
TComboBox — комбинированный список	310
TScrollBar — полоса прокрутки	311
TGroupBox — панель группирования	312
TRadioGroup — группа переключателей	312
TPanel — панель	313
TActionList — список действий	314
Глава 16. Компоненты вкладки Additional	317
TBitBtn — кнопка с изображением	317
TSpeedButton — кнопка панели инструментов	319
TMaskEdit — поле с маской ввода	320
TStringGrid — текстовая таблица	321
TDrawGrid — произвольная таблица	330
TImage — изображение	334
TShape — стандартная фигура	336
TBevel — кромка	337
TScrollBar — панель с полосами прокрутки	337
TCheckListBox — группа флажков	338
TSplitter — вешка разбивки	339
TStaticText — текстовая метка	340
TControlBar — контейнер для панели инструментов	341
TApplicationEvents — обработчик сообщений Windows	343
TValueListEditor — специализированный редактор списков	344
TLabelEdit — однострочное поле с меткой	348
TColorBox — список выбора цвета	348
TChart — диаграмма	349
TActionManager — менеджер действий	350
Глава 17. Компоненты вкладки Win32	353
TTabControl — набор вкладок	353
TPageControl — набор страниц с вкладками	355
TImageList — хранилище изображений	356
TRichEdit — поле формата RTF	356
TTrackBar — ползунок	360
TProgressBar — индикатор процесса	360
TUpDown — счетчик	361
TAnimate — анимация	362
TDateTimePicker — ввод и отображение даты/времени	366
TMonthCalendar — календарь	368

TTreeView — иерархическое дерево	369
TListView — иерархический список	378
THeaderControl — управляющий заголовок	383
TStatusBar — строка состояния	387
TToolBar и TToolButton — панель инструментов и кнопки для нее	389
TCoolBar и TCoolBand — панель инструментов и полосы для нее	392
TPageScroller — панель с кнопками прокрутки	394
TComboBoxEx — комбинированный список с расширенными возможностями	395
Глава 18. Компоненты вкладки System	400
TTimer — таймер	400
TPaintBox — окно для рисования	403
TMediaPlayer — медиаплеер	404
TOleContainer — контейнер OLE-объекта	413
Глава 19. Компоненты вкладки Dialogs	419
TOpenDialog и TSaveDialog — окна открытия и сохранения файлов	420
TOpenPictureDialog и TSavePictureDialog — окна открытия и сохранения изображений	423
TFontDialog — окно выбора шрифта	424
TColorDialog — окно выбора цвета	426
TPrintDialog — окно настройки параметров печати	426
TPrinterSetupDialog — окно настройки параметров принтера	428
TFindDialog — окно поиска	428
TReplaceDialog — окно поиска и замены	430
TPageSetupDialog — окно установки параметров печатаемой страницы	431
Глава 20. Компоненты вкладки Samples	432
TGauge — индикатор величины	432
TColorGrid — цветовая таблица	433
TSpinButton — спаренные кнопки	434
TSpinEdit — поле со спаренными кнопками	434
TDirectoryOutLine — структура каталогов	435
TCalendar — календарь	436
Компоненты Windows Shell	437
Глава 21. Форма	439
Разновидности форм	439
Компонент TForm	442
Создание и использование форм	450

Глава 22. Программа	454
Программные файлы	454
Файл проекта	454
Файлы модулей	458
Файлы ресурсов	459
Файлы настройки	459
Файлы резервных копий	460
Глобальные объекты	460
Application	460
Screen	466
Printer	468
Clipboard	473
Настройка программы	475
Файлы инициализации	475
Системный реестр Windows	480
Программа и Windows	486
Процессы и потоки	486
Использование памяти	494
Сообщения Windows	495
Функции API	500
 Часть IV. РАСШИРЕННЫЕ ВОЗМОЖНОСТИ DELPHI	502
Глава 23. Создание и использование DLL	503
Реализация	504
Пример создания DLL	505
Статическая загрузка	507
Динамическая загрузка	509
Интерфейсный модуль	510
Включение в библиотеку форм	511
Особенности использования модулей в DLL	515
 Глава 24. Интерфейсы	516
Создание и использование интерфейсов	516
Описание интерфейса	516
Идентификатор интерфейса	517
Компонентный класс	518
Класс TInterfacedObject и интерфейс IUnknow	519
Псевдонимы методов	520
Приведение интерфейсных типов	521
Уполномоченные свойства	521
Объекты автоматизации и интерфейс IDispatch	522

Глава 25. Технология COM	525
Некоторые детали технологии	525
Взаимодействие с помощью интерфейсов	526
Фабрика классов и автоматический запуск сервера	528
Библиотека типов	529
Пример использования	529
Сервер	530
Клиент	534
Использование вариантов в технологии OLE	535
Использование серверов пакета MS Office	538
Основные объекты серверов Excel и Word	539
Глава 26. Создание компонентов с помощью ModelMaker	542
Методика создания компонентов	542
Зачем нужны новые компоненты	543
Этапы разработки	543
Методика использования ModelMaker	554
Этапы разработки	554
Активизация ModelMaker	556
Вставка нового класса	556
Определение новых свойств и методов	556
Вставка исполняемого кода	557
Создание модуля с исходным кодом	559
Отладка компонента	562
Совершенствование компонента	562
Оформление компонента	564
Регистрация компонента	566
Приложение А. Интегрированная среда разработчика	568
Главное меню	568
Меню File	568
Меню Edit	571
Меню Search	573
Меню View	573
Меню Project	578
Меню Run	583
Меню Component	583
Меню Database	584
Меню Tools	584
Меню Help	591

Работа с редактором	595
Команды перемещения курсора	595
Команды удаления и вставки	596
Команды для работы с блоками	596
Прочие команды	597
Интеллектуальные возможности редактора	598
Отладка программ	600
Точки контрольного останова	600
Окно наблюдения	602
Принудительное прерывание работы программы	603
Трассировка программы	603
Действия в точках прерывания	603
Группировка точек прерывания	604
Вычисление выражений и изменение значений	605
Ведение протокола работы программы	606
Приложение Б. Некоторые стандартные подпрограммы, переменные, классы	607
Подпрограммы модуля Match	607
Подпрограммы модуля DateUtils	612
Спецификаторы формата даты/времени	617
Системные переменные, управляющие стандартным отображением данных	618
Правила использования параметров функции FloatToStrF	619
Спецификаторы форматирования вещественных чисел	620
Подпрограммы для работы с файлами	620
Стандартные классы исключений	622
Литература	628
Алфавитный указатель	629

От автора

Предлагаемая вашему вниманию книга содержит все необходимое для того, чтобы вы самостоятельно или под руководством преподавателя научились пользоваться популярнейшей в мире системой программирования Delphi, разработанной корпорацией Borland (США). Эта корпорация славится своими системами программирования для MS-DOS (Turbo Pascal, Turbo C, Turbo Assembler), Windows (Delphi, C++ Builder, JBuilder), Linux (Kylix).

Книга посвящена новейшей на сегодня (январь 2003 г.) версии Delphi 7 Studio и состоит из четырех частей.

В части I вы познакомитесь с основными приемами программирования в среде Delphi, а также с ее главными составными частями — галереей компонентов, хранилищем объектов, вспомогательным инструментарием. Эта часть фактически есть не что иное, как неформальное знакомство с Delphi.

В части II описывается язык программирования Delphi. Прочное освоение этой части гарантирует вам успешное решение с помощью Delphi любых задач.

Часть III посвящена компонентам — основным строительным «кирпичикам», из которых строится программа. В этой части описываются только компоненты общего назначения, которые пригодятся в любой программе. Многие специфические компоненты (например, компоненты, представленные на вкладках Decision Cube, ActiveX, ADO и другие галереи компонентов) остались вне рамок книги.

В части IV рассматриваются некоторые дополнительные возможности Delphi — динамически подключаемые библиотеки, интерфейсы, технология COM и система ModelMaker.

Вне рамок книги остались темы программирования баз данных, создания программ, использующих возможности Интернета, построения распределенных систем. В разделе «Литература», находящемся в конце книги, вы найдете перечень книг, в которых описываются эти возможности Delphi.

20 января 2003 г.

Введение

Это небольшое введение призвано дать самую общую характеристику системы программирования Delphi и оценить особенности ее версии 7 Studio.

Назначение и возможности Delphi

Delphi представляет собой систему программирования. Как любая подобная система, Delphi предназначена для разработки программ и имеет две характерные особенности: создаваемые с ее помощью программы могут работать не только под управлением Windows, а сама она относится к классу инструментальных средств ускоренной разработки программ (Rapid Application Development, RAD).

ПРИМЕЧАНИЕ

Есть вариант Delphi для работы под управлением ОС MVT IBM, а также версия для Linux (она называется Kylix). Версии 6 и 7 способны создавать кроссплатформенные приложения, то есть программы, могущие работать как под управлением Windows, так и под управлением Linux (на уровне исходных текстов; сама среда Delphi 6 или 7 и создаваемые ею программы работают только под Windows). Кроме того, в версию 7 Studio включена поддержка новейшей технологии Microsoft® .NET™, с помощью которой программу, созданную в Delphi, можно передать по Интернету любому абоненту и, если он использует ту же технологию, выполнить на его компьютере независимо от его программно-аппаратных особенностей (то есть под управлением любой операционной системы). Эта технология в книге не рассматривается.

Инструмент ускоренной разработки программ

Первый инструмент RAD создан корпорацией Microsoft и называется Visual Basic. Среда Delphi, созданная тремя годами позже, вместе с такими продуктами как Visual C++, C++ Builder, JBuilder, PowerBuilder также относится к классу инструментов ускоренной разработки программ. Это ускорение достигается за счет двух характерных свойств Delphi: визуального конструирования форм и широкого использования библиотеки визуальных компонентов (Visual Component Library, VCL).

Визуальное конструирование форм избавляет программиста от многих аспектов разработки интерфейса программы, так как Delphi автоматически готовит не-

обходимые программные заготовки и соответствующий файл ресурсов. Программист использует специальное окно, которое называется окном формы, как прототип будущего окна программы и наполняет его компонентами, реализующими нужные интерфейсные свойства (разного рода списки, кнопки, полосы прокрутки и т. п.). После размещения на форме очередного компонента Delphi автоматически вставляет в связанный с формой модуль ссылку на компонент и корректирует специальный файл описания формы с расширением DFM, который после компиляции преобразуется в ресурсный файл Windows.

Библиотека визуальных компонентов¹ предоставляет программисту огромное разнообразие созданных разработчиками Delphi программных заготовок, которые немедленно или после несложной настройки готовы к работе в рамках вашей программы. Компоненты характеризуются важным свойством: они включают в себя программный код и все необходимые для его работы данные, что избавляет программиста от рутинной работы по «изобретению велосипедов» — нет нужды писать то, что уже написано (и в подавляющем большинстве — очень грамотно!) другими. Как уже упоминалось, с Delphi поставляется великое множество компонентов, рассчитанных на самые разные аспекты применения — от простеньких компонентов, создающих поясняющие надписи, до сложных текстовых процессоров или инструментов принятия решений. Если по каким-либо причинам в Delphi нет компонента с нужной функциональностью, его можно создать средствами самой среды Delphi и включить затем в VCL (другой вариант — обратиться к ресурсам Интернета, где на тысячах сайтов предлагаются бесплатные, условно-бесплатные и платные компоненты, созданные специально для Delphi как профессиональными программистами, так и любителями).

Использование компонентов не только во много раз сокращает сроки разработки программ, но и существенно снижает вероятность случайных программных ошибок, от которых, увы, не защищен ни один крупный программный проект. Однако в их применении есть и обратная сторона. Как правило, даже простые в функциональном отношении компоненты представляют собой лишь «вершины айсбергов», так как они создаются по объектно-ориентированной технологии и многие их функциональные черты наследуются от многочисленных родительских компонентов. В результате даже несложные программы, созданные в Delphi, редко имеют объем меньше сотен килобайт.

Ни один инструмент RAD не создается без средств прогона и отладки вновь созданной программы. Такими средствами традиционно оснащаются все инструменты, создаваемые корпорацией Borland. В Delphi эти средства доведены до совершенства. Вы можете начать исполнение программы и последовательно — оператор за оператором — следить за ее выполнением по исходному тексту. В любой момент вы можете узнать текущее значение переменной и при необходимости изменить его без перекомпиляции программы. Вы можете запустить программу, установив в ней нужные точки контрольного останова — при достижении такой точ-

¹ Несмотря на слова «визуальных компонентов» в названии библиотеки VCL, около трети входящих туда компонентов не имеют видимого воплощения в работающей программе (компоненты действий, хранилище изображений, таймер, компоненты связи с удаленными компьютерами, все источники данных и т. д.).

ки программа автоматически прервет свою работу и перейдет в отладочный режим. Точки останова можно делать условными и объединять их в группы, разрешая или запрещая их использование. В некоторых случаях нельзя непосредственно проследить за работой программы (например, обычно нельзя проследить за работой сервера, который автоматически запускается клиентом). В таких ситуациях можно заставить программу создать файл протокола работы, анализ которого затем позволит установить причину ошибки. Опытные программисты в трудных случаях могут использовать средства отладки на уровне машинных инструкций.

Мощность и гибкость языка

Мощность и гибкость языка программирования Delphi (в версиях 1...6 этот язык называется Object Pascal, однако в версии 7 он переименован в Delphi; с точки зрения семантики и синтаксиса языки Object Pascal v.6 и Delphi v.7 почти идентичны) — безусловное достоинство Delphi, выгодно отличающее эту систему программирования от других инструментов RAD. Ядром языка Delphi является язык Паскаль¹, созданный профессором Цюрихского университета Никлаусом Виртом еще в конце 60-х гг. специально для обучения студентов программированию. В числе его студентов были Филипп Канн (Kahn) и Андерс Хейлсберг (Hejlsberg). Первый основал корпорацию Borland, второй стал ее главным программистом. За долгие годы А. Хейлсберг и программисты под его началом² превратили Паскаль в мощный современный объектно-ориентированный язык, пригодный для эффективного создания программ любой сложности.

От Visual Basic язык Delphi отличают строгая типизированность, позволяющая компилятору еще на этапе компиляции обнаружить многие ошибки, а также средства работы с указателями. Последнее дает возможность использовать так называемое раннее связывание с библиотеками типов в технологии СОМ, в то время как Visual Basic (и Java, в котором тоже не поддерживаются указатели) вынуждены при обращении к СОМ использовать более медленное позднее связывание и интерфейсы диспетчеризации.

Синтаксис С++ прямо-таки провоцирует создание запутанных программ, в которых трудно разобраться даже автору, в то время как простой и ясный синтаксис Delphi позволяет последнему претендовать на роль языка, идеально подходящего для описания алгоритма (недаром Паскаль происходит от использующегося для этих целей алгоритмического языка АЛГОЛ-60).

Если по каким-либо причинам возможности Delphi окажутся недостаточными, вы можете программировать на Ассемблере (машинно-зависимом языке программирования), который органично вплетен в Delphi.

Во всех случаях Delphi имеет самый быстрый среди продуктов подобного рода оптимизирующий компилятор, позволяющий создавать быстрые и относительно компактные программы.

¹ Назван в честь выдающегося средневекового философа и математика Блеза Паскаля.

² Несколько лет назад (в 1997 г.) А. Хейлсберг перешел в корпорацию Microsoft, и его можно понять: вместо 200 000 долларов в год Б. Гейтс предложил ему 2 500 000. В настоящее время главным программистом Borland (и идеологом Delphi) является ученик Хейлсберга Чак Яджевски (Jazdzewski).

Инструмент создания приложений баз данных

Система Delphi завоевала себе репутацию самого эффективного средства разработки приложений баз данных, то есть программ, обслуживающих электронные хранилища информации. Эта репутация определяется тремя обстоятельствами: высокопроизводительной машиной доступа к данным разного формата (Borland Database Engine, BDE¹), наличием многочисленных компонентов и технологий, ориентированных на эту сферу применения, и поставкой вместе с Delphi компактного, мощного и простого в администрировании сервера баз данных InterBase.

Машина BDE является средством доступа к данным разного формата (локальным таблицам типа dBASE, Paradox, к серверам баз данных InterBase, MS SQL Server, Oracle, Informix и т. п.). Особенностью BDE является использование собственных высокопроизводительных драйверов для доступа к локальным таблицам dBASE, Paradox, FoxPro, а также SQL-ориентированных драйверов для доступа к серверам баз данных (этот набор драйверов называется SQL Links). Вместо собственных драйверов при желании могут использоваться драйверы сторонних производителей, соответствующие стандарту ODBC (Open DataBase Connectivity Interface — открытый интерфейс взаимодействия с базами данных). Более того, в версиях 6 и 7 Delphi появилась возможность использования альтернативных BDE средств доступа по технологиям ADO, InterBase Express, dbExpress.

Многочисленные компоненты, поддерживающие разработку приложений баз данных, обеспечивают решение самых разных задач: выборку и сортировку данных, их наглядное представление (в том числе и графическое), изменение и публикацию данных в виде отчетов (документов) и/или HTML-страниц в Интернете и т. д.

Поставляемый вместе с Delphi (в наиболее дорогостоящих вариантах поставки) сервер InterBase фактически стал известен в России именно благодаря Delphi (до появления первой версии Delphi его поставки в нашу страну были запрещены в связи с тем, что InterBase широко используется в различных системах армии США). Этот сервер создан дочерним отделением Borland — InterBase Software Corporation, а его поставка вместе с Delphi объясняется необходимостью дать разработчикам средство испытания программ, рассчитанных на архитектуру клиент-сервер. Сервер InterBase в максимальной степени соответствует стандарту SQL-92, которому в той или иной мере следуют все остальные серверы баз данных.

Особенности версии Delphi 7 Studio

Последняя на сегодня (начало 2003 г.) версия Delphi 7 Studio сохраняет почти полную преемственность по языку программирования и интегрированной среде разработчика с версией 6. Ее характерное отличие — введение множества новых технологий, облегчающих (и упрощающих) создание программ для баз данных и Интернета.

¹ Корпорация Borland официально объявила о возможном отказе от этого инструмента в своих будущих разработках.

ПРИМЕЧАНИЕ

В отличие от предыдущих версий, версия Delphi 7 Studio содержит минимальное устанавливаемое ядро. Все описываемые далее технологии не принадлежат этому ядру и требуют отдельной установки. В зависимости от установленных компонентов меняется главное меню системы. Дополнительные технологии в большинстве своем разработаны сторонними фирмами и поставляются на двух компакт-дисках. Более того, некоторые технологии работают только под управлением Windows NT/2000/XP, поэтому базовой платформой для развертывания этой версии Delphi должна быть одна из указанных операционных систем.

Речь идет, прежде всего, об активно разрабатываемой корпорацией Microsoft технологии .NET. Эта технология представляет собой совокупность стандартов и инструментов, позволяющих во многом упростить распространение программ по Интернету. В основе технологии лежит давно известная идея¹ создания некоего промежуточного языка программирования и среды выполнения этого языка. В Delphi включен компилятор языка .NET, с помощью которого любая программа заново кодируется и затем с использованием стандартных протоколов HTTP и SOAP пересылается по Интернету адресату. Если адресат имеет среду выполнения этого языка, он сможет воспроизвести эту программу независимо от своих программно-аппаратных особенностей. Таким образом, программа, созданная под управлением Windows, может исполняться под управлением Linux, ОС Mac, Solaris и т. д.

Инструмент CASE (Computing Assisted Software Engineering — средство автоматизированного программирования) под названием ModelMaker позволяет разработчикам программ сосредоточить свои усилия на творческой стороне процесса программирования, оставляя всю черновую работу ModelMaker. С помощью ModelMaker существенно упрощается создание любых сущностей проекта (классов, интерфейсов, модулей и т. д. — вплоть до практически готовой службы встроеной помощи).

Существенно изменены технологии создания приложений для Интернета. С помощью нового набора компонентов IntraWeb можно готовить приложения, которые не будут требовать обязательного посредничества веб-сервера между созданным вами приложением и браузером пользователя (только для режима Stand Alone Application; для остальных типов приложений наличие промышленных серверов обязательно). Введена поддержка промышленного сервера Apache 2.

Устранены любые отличия в версиях языка Delphi для собственно Delphi 7 и Kylix 3, что в еще большей степени упрощает процесс создания кросс-платформенных приложений. Однако попытка унификации VCL и CLX привела к тому, что из VCL исчезли компоненты QuickReport и FastNet, которые следует заменять компонентами вкладок Rave и Indy.

Замечу, что детальный анализ новых технологий выходит за рамки темы данной книги.

¹ Впервые эта идея была высказана еще в 1958 г. американцем Мелвином Е. Конвеем (Conway) в журнальной статье «Proposal For An UNCOL» («Предложение по универсальному компьютерно-ориентированному языку»).

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы сможете найти по адресу <http://www.piter.com/download>.

Подробную информацию о наших книгах вы найдете на веб-сайте издательства <http://www.piter.com>.

Часть I

Знакомство с Delphi

Главы этой части предназначены для неформального знакомства с системой программирования Delphi. Они дадут вам самое общее представление о среде разработчика, языке Delphi и основных приемах визуального программирования

Глава 1

Среда разработчика

Интегрированная среда разработчика (ИСП) Delphi — это сложный механизм, обеспечивающий высокоэффективную работу программиста. Визуально она реализуется несколькими одновременно раскрытыми на экране окнами. Окна могут перемещаться по экрану, частично или полностью перекрывая друг друга, что обычно вызывает у пользователя, привыкшего к относительной «строгости» среды текстового процессора Word или табличного процессора Excel, ощущение некоторого дискомфорта. После приобретения опыта работы с Delphi это ощущение пройдет, и вы научитесь быстро отыскивать нужное окно, чтобы изменить те или иные функциональные свойства создаваемой вами программы, ибо каждое окно несет в себе некоторую *функциональность*, то есть предназначено для решения определенных задач.

Запустите Delphi — и вы увидите нечто похожее на рис. 1.1 с шестью наиболее важными окнами Delphi.

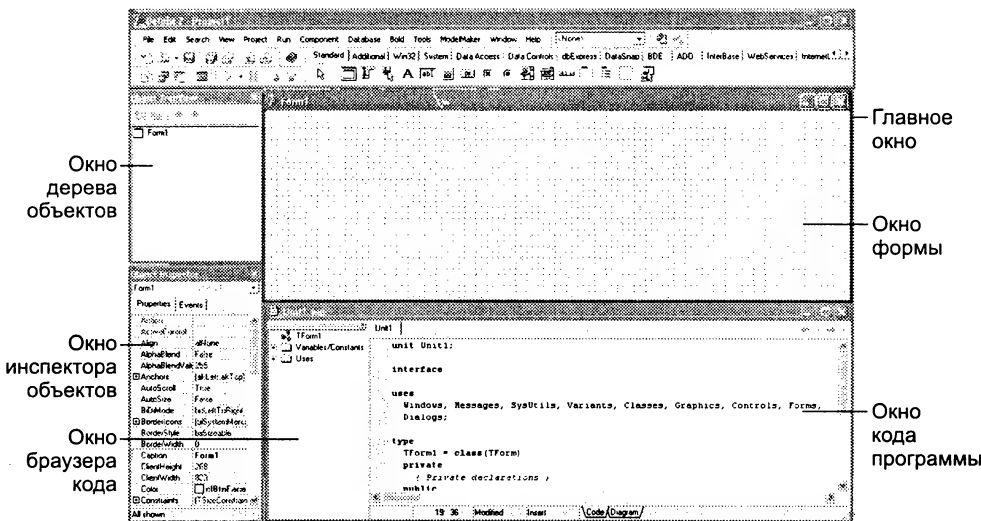


Рис. 1.1. Наиболее важные окна Delphi

Чтобы упорядочить окна так, как показано на рисунке, вам придется вручную изменить их положение и размеры, так как обычно окно кода программы почти полностью перекрыто окном формы. Впрочем, добиваться максимального сходства того, что вы видите на экране вашего компьютера, с изображением, показанным на рис. 1.1, вовсе необязательно: расположение и размеры окон никак не влияют на их функциональность; если вы не видите на экране окна кода, но хотите его увидеть, просто нажмите клавишу F12¹. Повторное нажатие клавиши F12 вновь активизирует окно формы и т. д. — эта клавиша работает как двоичный переключатель, попеременно показывая окно формы или окно кода программы.

Замечу, что если в системе установлена технология Bold, при первом запуске Delphi поверх всех окон появится окно, показанное на рис. 1.2.

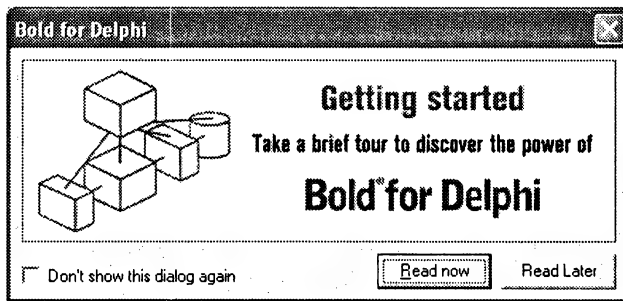


Рис. 1.2. Окно технологии Bold

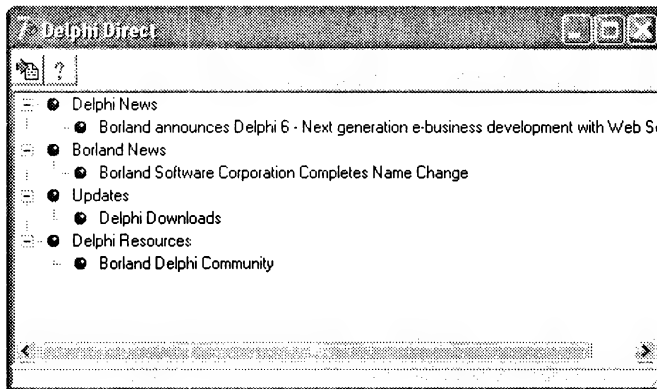


Рис. 1.3. Окно Delphi Direct

С помощью этого окна вы сможете получить доступ к справочной службе и веб-сайту шведской фирмы BoldSoft MDE AB, которая разработала эту технологию. С периодичностью, определяемой элементами управления вкладки Delphi Direct диалогового окна Environment Options (открывается командой Tools ► Environment Options), появляется окно Delphi Direct (рис. 1.3), предназначенное для просмотра самой све-

¹ Если к этому моменту активным было окно инспектора объектов, вам придется нажать клавишу F12 еще раз.

жей информации о корпорации и ее программных продуктах, копирования дополнительных файлов, чтения ответов на наиболее часто задаваемые вопросы и т. д. Вы можете вызвать это окно из главного окна Delphi в любой момент с помощью команды Help ► Delphi Direct.

Главное окно

Главное окно осуществляет основные функции управления проектом создаваемой программы. Это окно всегда присутствует на экране и упрямо занимает его самую верхнюю часть. Не пытайтесь его распахнуть на весь экран: даже в развернутом состоянии его размеры и положение практически не отличаются от обычных.

Связано это с функциональностью главного окна: с одной стороны, оно несет в себе элементы, которые всегда должны быть «под рукой» у программиста, с другой, — окно не должно отнимать у остальных окон Delphi значительного пространства экрана. Свертывание главного окна приводит к исчезновению с экрана других окон Delphi¹, а его закрытие означает окончание работы программиста с системой программирования.

В главном окне располагается главное меню Delphi, набор инструментальных кнопок и палитра компонентов.

Главное меню² содержит все необходимые средства для управления проектом. Все пункты главного меню открывают доступ к меню второго уровня (подменю). Для начального знакомства с Delphi мы вполне обойдемся инструментальными кнопками и нам не понадобится обращаться к главному меню, вот почему подробное описание его команд вы найдете не здесь, а в приложении А.

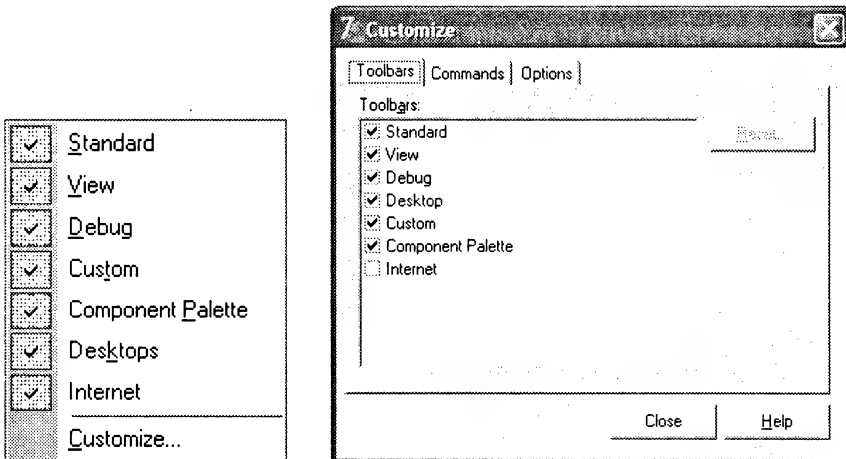


Рис. 1.4. Настройка инструментальных панелей: слева — контекстное меню; справа — окно настройки

¹ Эти окна появятся, как только главное окно будет восстановлено в размерах.

² Помимо главного меню в Delphi широко используется система контекстных меню, доступ к которым осуществляется по щелчку правой кнопкой мыши.

Все элементы главного окна располагаются на специальных панелях, в левой или верхней части которых имеются вешки перемещения, позволяющие с помощью мыши перетаскивать панели вместе с находящимися на них элементами. Любую панель (кроме главного меню) можно убрать из окна (сделать ее невидимой) или «пустить плавать» по экрану в отдельном окне. Для этого нужно лишь «стасщить» панель с помощью мыши за вешку из пределов главного окна. Для изменения состава представленных на панели кнопок нужно предварительно щелкнуть на ней правой кнопкой мыши. В появившемся после этого контекстном меню (рис. 1.4, слева) перечислены названия всех панелей и указан их статус (отмеченные флажками панели видны в главном окне; если флажок снять, панель исчезнет). После выбора команды *Customize* (Настройка) появляется окно настройки (рис. 1.4, справа). Теперь можно «стаскивать» с панелей ненужные кнопки или, перейдя на вкладку *Commands*, выбирать в списке названия нужных кнопок и перетаскивать их на экран.

Для примера на рис. 1.5 показаны элементы главного окна с нестандартным расположением панелей и представленных на них кнопок. При его компоновке мне хотелось, с одной стороны, получить больше места для размещения страниц галереи компонентов, а с другой — поместить в главное окно кнопки часто используемых мною команд меню:

-  — *Project* ▶ *Build Project* (полная перекомпиляция проекта);
-  — *Project* ▶ *Syntax Check* (проверка синтаксиса кода программы);
-  — *View* ▶ *Debug Windows* ▶ *Watches* (доступ к окну наблюдения отладочного режима);
-  — *View* ▶ *Debug Windows* ▶ *Breakpoints* (доступ к окну точек останова).



Рис. 1.5. Нестандартное расположение элементов в главном окне

Инструментальные кнопки

Инструментальные кнопки открывают быстрый доступ к наиболее важным командам главного меню. По функциональному признаку они разделены на 7 групп. Каждая группа занимает отдельную панель. В табл. 1.1 описаны команды, реализуемые стандартным набором кнопок.

Таблица 1.1. Стандартный набор кнопок панелей инструментов Delphi

Кнопка	Описание
<i>Группа Standard</i>	
	Открывает доступ к хранилищу объектов. Эквивалент команды <i>File</i> ▶ <i>New</i> ▶ <i>Other</i>

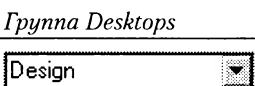
Кнопка	Описание
	Открывает существующий файл. Эквивалент команды File ► Open File
	Сохраняет файл на диске. Эквивалент команды File ► Save File (клавиши быстрого доступа — Ctrl+S)
	Сохраняет все файлы проекта. Эквивалент команды File ► Save All
	Открывает созданный ранее проект программы. Эквивалент команды File ► Open Project (клавиши быстрого доступа — Ctrl+F11)
	Добавляет новый файл к проекту. Эквивалент команды Project ► Add to Project (клавиши быстрого доступа — Shift+F11)
	Удаляет файл из проекта. Эквивалент команды — Project ► Remove from Project
<i>Группа View</i>	
	Выбирает модуль из списка модулей, связанных с текущим проектом. Эквивалент команды View ► Units (клавиши быстрого доступа — Shift+F12)
	Выбирает форму из списка форм, связанных с текущим проектом. Эквивалент команды View ► Forms (клавиши быстрого доступа — Ctrl+F12)
	Переключает активность между окном формы и окном кода программы. Эквивалент команды View ► Toggle Form/Unit (клавиша быстрого доступа — F12)
	Создает новую форму и добавляет ее к проекту. Эквивалент команды File ► New ► Form
<i>Группа Debug</i>	
	Компилирует и выполняет программу. Эквивалент команды Run ► Run (клавиша быстрого доступа — F9)
	Приостанавливает выполнение отлаживаемой программы. Эквивалент команды Run ► Program Pause
	Осуществляет пошаговую трассировку программы с вызываемыми подпрограммами. Эквивалент команды Run ► Trace Into (клавиша быстрого доступа — F7)
	Осуществляет пошаговую трассировку программы без трассировки вызываемых подпрограмм. Эквивалент команды Run ► Step Over (клавиша быстрого доступа — F8)
<i>Группа Custom</i>	
	Открывает доступ к встроенной справочной службе. Эквивалент команды Help ► Delphi Help
<i>Группа Desktops</i>	
	Список для выбора возможных вариантов настройки остальных окон Delphi

Таблица 1.1 (продолжение)

Кнопка	Описание
	Сохраняет текущую настройку окон Delphi
	Выбирает настройку окон, соответствующую отладочному режиму
<i>Группа Internet</i>	
	Начинает создание нового приложения по технологии WebSnap (приложения для Интернета)
	Создает новую страницу приложения WebSnap
	Создаст новый модуль данных приложения WebSnap

Настройка окон

Начиная с версии 5 главное окно обогатилось тремя новыми интерфейсными элементами, перечисленными в группе Desktops табл. 1.1. С помощью этих инструментов программист может подготовить несколько вариантов расположения остальных окон Delphi и сохранить их в настроечном файле.

Обычно выбирается две или три основные конфигурации окон: для режима разработки форм, для кодирования и для отладки. При разработке формы на экране должна быть видна сама форма, дерево объектов и инспектор объектов. Настроив соответствующие размеры и положение этих окон, такую настройку можно сохранить под именем, например, DesignDesk. Для режима кодирования обычно нужно только максимально раскрытое окно кода с браузером кода и пристыкованное к нему окно дерева объектов. Такую конфигурацию можно сохранить под именем CodeDesk. Наконец, в отладочном режиме к окну кода желательно «прицепить» некоторые отладочные окна, такие как Watches (окно наблюдений за переменными/выражениями) и Breakpoints (окно создания/настройки точек контрольного останова). Такую конфигурацию можно сохранить под именем DebuggDesk, например. Теперь при переходе к тому или иному режиму достаточно раскрыть список и выбрать в нем имя нужной конфигурации. Если вы предварительно с помощью кнопки  настроите конфигурацию для отладочного режима, эта конфигурация появится автоматически, как только среда перейдет в режим отладки.

Палитра компонентов

Палитра компонентов — это главное богатство Delphi. Она занимает правую часть главного окна и имеет вкладки, обеспечивающие быстрый поиск нужного компонента. Под компонентом понимается некий функциональный элемент, содержащий определенные свойства и размещаемый программистом в окне формы. С помощью компонентов создается каркас программы, во всяком случае, — ее видимые на экране внешние проявления: окна, кнопки, списки и т. д.

Как и инструментальная панель, палитра компонентов может настраиваться. Для этого используется специальный редактор (рис. 1.6), окно которого появляется

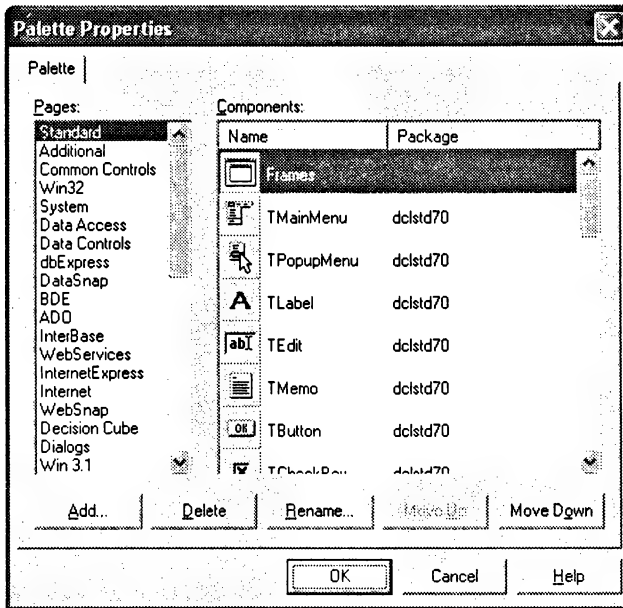


Рис. 1.6. Окно редактора палитры компонентов

на экране после щелчка правой кнопкой мыши на любой кнопке в палитре компонентов и выбора команды Properties (Свойства).

ПРИМЕЧАНИЕ Содержимое вкладок (страниц) палитры компонентов в Delphi 7 зависит от целевой платформы создаваемого приложения (Windows или Linux), а также от того, что в данный момент присутствует в окне формы (собственно форма, модуль данных и т. д.).

Нет смысла удалять какой-то из компонентов, а создавать новые мы с вами пока еще не умеем, поэтому я не рекомендую вам экспериментировать с редактором палитры. Тем не менее, один совет все-таки дам. В стандартном наборе вкладок палитры компонентов есть вкладка Samples, содержащая часто используемые компоненты. В то же время ее ярлычок в палитре закрыт двумя небольшими кнопками «прокрутки» вкладок палитры. Имеет смысл поменять ее местами с менее используемой вкладкой, например, с ADO. Вы можете «перетащить» пункт Samples в верхнюю часть списка Pages редактора палитры, «положив» его на пункт ADO. После такой перестановки ярлычок Samples станет доступным в любой момент. Если, к тому же, вы не планируете использовать в своих проектах технологию ADO для доступа к базам данных, перетащите пункт ADO в самый конец списка.

Окно формы

Окно формы представляет собой проект окна будущей программы. Вначале это окно пусто. Точнее, оно содержит стандартные для Windows интерфейсные элементы — кнопки вызова системного меню, развертывания, свертывания и закры-

тия окна, строку заголовка и габаритную рамку. Вся рабочая область окна обычно заполнена точками координатной сетки, служащей для упорядочения размещаемых на форме компонентов (эти точки отображаются только на этапе конструирования программы; вы можете их убрать, вызвав с помощью команды Tools ► Environment Options соответствующее окно настроек и сняв флажок Display Grid на вкладке Designer).

Значительную часть времени программист занят увлекательным занятием, напоминающим работу с набором деталей конструктора Lego: он «достает» из палитры компонентов, как из коробки с деталями, нужный компонент и размещает его на «наборном поле» окна формы, постепенно заполняя форму интерфейсными элементами. Собственно, именно в этом процессе наполнения формы и заключается главная изюминка визуального программирования. Программист в любой момент времени контролирует вид окна создаваемой программы и может внести в него необходимые изменения.

Окно дерева объектов

Это окно предназначено для наглядного отображения связей между отдельными компонентами, размещенными на активной форме или в активном модуле данных. Щелчок на любом компоненте в этом окне активизирует соответствующий компонент в окне формы и отображает свойства этого компонента в окне инспектора объектов. Двойной щелчок приводит к срабатыванию механизма Code Insight, который вставляет в окно кода заготовку для обработчика события OnClick. Наконец, компонент можно «перетащить» в окне на другое место и таким образом поменять его владельца (свойство Parent — родитель). В предыдущих версиях такую замену можно сделать только с помощью межпрограммного буфера обмена (clipboard).

Окно инспектора объектов

Любой размещаемый на форме компонент характеризуется некоторым набором параметров: положением, размером, цветом и т. д. Часть этих параметров, например, положение и размеры компонента, программист может изменять, манипулируя компонентом в окне формы. Для изменения других параметров предназначено окно инспектора объектов. Это окно содержит две вкладки — Properties (Свойства) и Events (События). Вкладка Properties служит для установки нужных свойств компонента, вкладка Events позволяет определить реакцию компонента на то или иное событие. Совокупность свойств отображает видимую сторону компонента: положение относительно левого верхнего угла рабочей области формы, его размеры и цвет, шрифт и текст надписи на нем и т. п.; совокупность событий — его поведенческую сторону: будет ли компонент реагировать на щелчок мыши или на нажатие клавиш, как он будет вести себя в момент появления на экране или в момент изменения размеров окна и т. п.

Каждая вкладка окна инспектора объектов представляет собой таблицу из двух колонок: левая колонка содержит название свойства или события, а правая — кон-

кретное значение свойства или имя подпрограммы¹, обрабатывающей соответствующее событие.

Строки таблицы выбираются щелчком мыши, и в них могут отображаться простые или сложные свойства. К простым относятся свойства, определяемые единственным значением — числом, строкой символов, значением True (Истина) или False (Ложь) и т. п. Например, свойство Caption (Заголовок) представляется строкой символов, свойства Height (Высота) и Width (Ширина) — числами, свойство Enabled (Доступность) — значениями True или False. Сложные свойства определяются совокупностью значений. Слева от имени таких свойств указывается значок «+», а щелчок мышью на нем приводит к раскрытию списка составляющих сложного свойства. Чтобы закрыть раскрытый список, нужно щелкнуть на значке «-», в который после щелчка превращается значок «+».

Щелчок мышью на правой колонке любой строки таблицы активизирует указанное в ней значение свойства, при этом в правом конце строки может появиться кнопка  или . Щелчок на кнопке  приводит к появлению на экране диалогового окна, с помощью которого устанавливается значение сложного свойства. Щелчок на кнопке  ведет к раскрытию списка возможных значений простого свойства.

В верхней части окна инспектора объектов располагается раскрывающийся список всех помещенных на форму компонентов. Поскольку форма сама по себе является компонентом, ее имя также присутствует в этом списке. В отличие от предыдущих версий, раскрывающийся список версий 6 и 7 содержит не только имена компонентов, но и названия их классов (подробнее о классах мы поговорим в главе 8).

В контекстном меню окна, которое появляется после щелчка на нем правой кнопкой мыши, имеется ряд команд, позволяющих настроить окно. В частности, после выбора команды Stay on Top окно инспектора объектов будет «плавать» над всеми другими окнами независимо от его активности. Такое состояние окна удобно при частом его использовании, например, при конструировании сложной формы, содержащей множество компонентов. Если выбрать в контекстном меню команду Arrange ► By Category, все строки окна инспектора объектов будут представлять собой раскрывающиеся списки свойств и событий, упорядоченные по категориям (рис. 1.7).

Любые категории можно скрыть. Для этого нужно в контекстном меню раскрыть вложенное меню View и убрать флажок слева от имени категории.

С помощью команды Properties контекстного меню окна инспектора объектов можно отменить возможность представления в этом окне сложных свойств. Для этого в списке SpeedSettings на вкладке Object Inspector соответствующего диалогового окна нужно выбрать пункт Delphi 5 emulation (Эмуляция Delphi 5).

Поэкспериментируйте самостоятельно с контекстным меню инспектора объектов, чтобы понять назначение тех или иных его команд. Если вы случайно или намеренно сделаете окно невидимым, нажмите клавишу F11 или выберите команду View ► Object Inspector, чтобы оно вновь появилось на экране.

¹ Если вам еще не знаком этот термин, считайте, что подпрограмма — это просто относительно небольшой фрагмент программы.

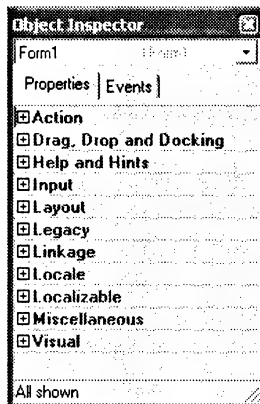


Рис. 1.7. Списки свойств, упорядоченные по категориям

Окно кода программы

Окно кода предназначено для создания и редактирования текста программы. Этот текст составляется по специальным правилам и описывает алгоритм работы программы. Совокупность правил записи текста называется языком программирования. В системе Delphi используется язык программирования Delphi, который представляет собой значительно расширенную версию широко распространенного языка Паскаль, впервые предложенного швейцарским профессором Н. Виртом еще в 1970 г. и усовершенствованного сотрудниками корпорации Borland (созданные ими языки назывались Turbo Pascal, Borland Pascal и Object Pascal; последний в версии 7 переименован в Delphi). Несмотря на то, что визуальная среда Delphi берет на себя многие рутинные аспекты программирования, знание языка Delphi является неременным условием для любого программиста, работающего с этой системой.

Первоначально окно кода содержит минимальный исходный текст, обеспечивающий нормальное функционирование пустой формы в качестве полноценного окна Windows-программы. В ходе работы над проектом программист вносит в него необходимые дополнения, чтобы придать программе нужную функциональность. Поскольку для создания даже простых программ вам понадобится создавать и изменять (редактировать) код программы, ниже описываются основные приемы работы с окном кода.

Сразу после открытия нового проекта в нем будут строки, показанные в листинге 1.1.

Листинг 1.1. Код для пустой формы¹

```
unit Unit1;

interface
```

¹ Все представленные в книге тексты программ можно загрузить с сайта издательства по адресу <http://www.piter.com/download>. — Примеч. ред.

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs;

type

```
TForm1 = class (TForm)
private
    { Private declarations }
public
    { Public declarations }
end;
```

var

```
Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

end.

Эти строки Delphi автоматически вставляет в окно кода для новой формы. Как уже говорилось, окно кода определяет поведенческую сторону окна программы (то есть окна, появляющегося после начала работы программы), а окно формы — его внешние проявления. Оба окна тесно связаны друг с другом, причем Delphi «хозяйничает» в его верхней части, вставляя необходимые строки между строками **unit** `Unit1` и **implementation**.

Пока не стоит изменять эту часть текста. В дальнейшем мы будем вставлять в окно текст программы между строками `{ $R *.DFM }` и **end**.

Чтобы вставить в окно новую строку (строки), нужно сначала с помощью клавиш управления курсором или щелчком мыши установить текстовый курсор (мигающую вертикальную черту) в нужное место, а затем ввести текст с клавиатуры. Обычно текст кода программы располагается в нескольких строках. Для перехода на новую строку используйте клавишу Enter.

Если в процессе ввода вы ошиблись и тут же заметили свою ошибку, удалите ошибочный символ клавишей Backspace. Клавиша Backspace удаляет символ *слева* от курсора, а клавиша Delete — *справа* от него. Если понадобится удалить сразу всю строку текста, установите курсор в любое место строки, нажмите клавишу Ctrl и, не отпуская ее, клавишу Y. Такое совместное нажатие клавиш в дальнейшем будем обозначать символами Ctrl+Y. Чтобы отменить последнее изменение текста, нажмите клавиши Ctrl+Z или выберите команду Edit ► Undo.

Более полное руководство по работе с текстовым редактором окна кода приводится в приложении А.

Вместе с окном кода обычно активизируется также и окно браузера кода, облегчающее поиск нужных элементов в случае, когда в окне набрано много строк кода (рис. 1.8).

СОВЕТ Текстовый редактор Delphi версии 7 имеет расширенные «интеллектуальные» возможности Code Insight, связанные с контролем текста по мере его ввода программистом. К сожалению, этот контроль отнимает у компьютера определенные ресурсы. Чтобы избавиться от раздражающих пауз в процессе набора кода, особенно заметных на компьютерах с памятью менее 32 Мбайт, оснащенных процессорами с тактовой частотой 100 МГц и менее, выберите в главном меню команду Tools ► Environment Options, в открывшемся окне перейдите на вкладку Code Insight и снимите флажки Code Completion, Code Parameters и Tooltip Symbol Insight. Увы! Вы лишитесь чудесных возможностей режима Code Insight, но зато обретете полный контроль над клавиатурой. Флажок Tooltip Expression Evaluation разрешает/запрещает вычисление и показ значения переменной в режиме отладки, когда над переменной останавливается указатель мыши. Снимать этот флажок не имеет смысла, так как на этапе ввода текста программы этот сервис среды заблокирован и активизируется лишь после перехода в режим отладки.

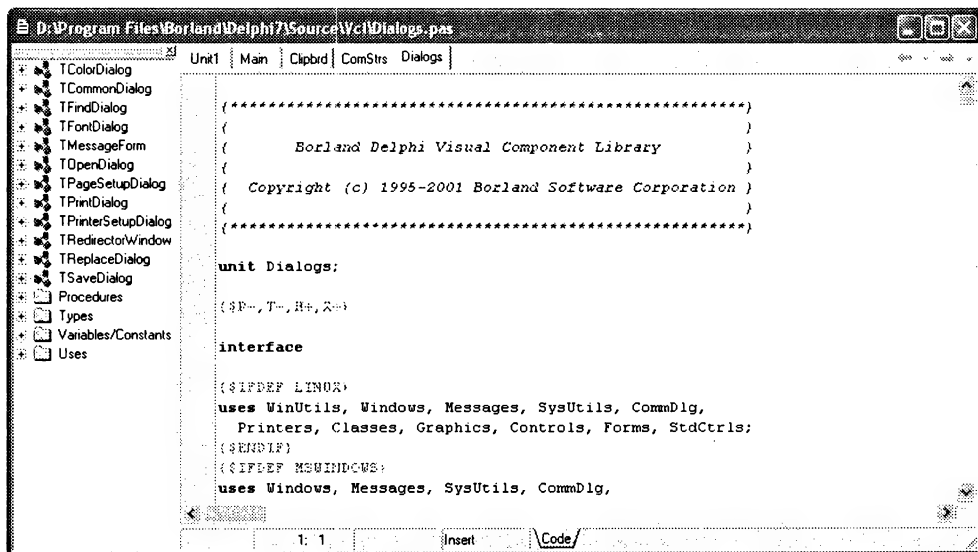


Рис. 1.8. Окно кода с расположенным слева окном браузера кода

Двойной щелчок мышью на элементе в окне браузера приводит к позиционированию текстового курсора в окне кода на описание соответствующего объекта или на строку с его первым упоминанием. Окно браузера можно закрыть щелчком на кнопке  в его верхнем правом углу. Восстановить закрытое окно можно щелчком правой кнопки мыши в окне кода и последующим выбором команды View Explorer в контекстном меню. Замечу, что окно браузера располагается на вспомогательной панели и имеет вешку перемещения  в верхней части окна. «Схватив» ее мышью, вы можете перетащить окно на любое свободное место экрана или «причалить» его к окну инспектора объектов.

В версиях 6 и 7 на нижней кромке окна кода располагаются ярлычки двух вкладок — Code и Diagram. Щелчок на ярлычке Diagram активизирует вкладку диаграмм

(рис. 1.9). Первоначально эта вкладка пуста. Чтобы наполнить ее, нужно перетащить мышью два и более компонентов из окна дерева объектов (на рисунке это окно пристыковано к верхней части окна браузера). Чтобы перетащить компоненты на вкладку диаграмм, нужно нажать клавишу Shift и, удерживая ее, выполнить щелчки мышью на нужных компонентах (выбрать их), затем отпустить клавишу Shift и перетащить мышью выбранные компоненты на вкладку диаграмм.

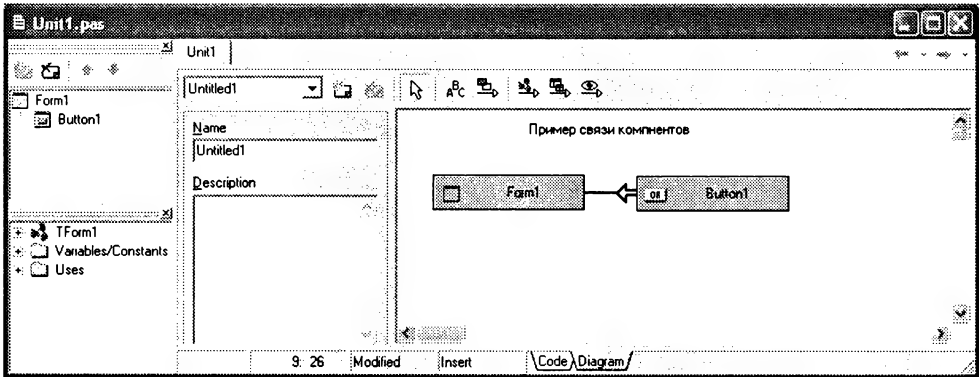


Рис. 1.9. Пример страницы диаграмм

Для сложных форм с множеством компонентов можно подготовить несколько диаграмм. Дав каждой диаграмме уникальное имя, можно в любой момент с помощью раскрывающегося списка выбрать нужную.

Вкладку диаграмм можно напечатать.

Глава 2

Основы визуального программирования

Программирование в Delphi строится на тесном взаимодействии двух процессов: процесса конструирования интерфейса, или визуального проявления, программы (то есть ее окна) и процесса написания кода, придающего элементам этого окна и программе в целом необходимую функциональность. Для написания кода используется окно кода, для конструирования программы — остальные окна Delphi и, прежде всего, окно формы.

Между содержимым окон формы и кода существует неразрывная связь, которая строго отслеживается Delphi. Это означает, что размещение на форме компонента приводит к автоматическому изменению кода программы и наоборот — удаление тех или иных автоматически вставленных фрагментов кода может привести к удалению соответствующих компонентов. Помня об этом, программисты вначале конструируют форму, размещая на ней очередной компонент, и только после этого переходят, если это необходимо, к написанию кода, обеспечивающего требуемое поведение компонента в работающей программе. Вот почему наше знакомство с программированием в Delphi мы начинаем с окна формы.

Пустая форма и ее модификация

Как уже говорилось, окно формы содержит проект окна программы. Важно помнить, что с самого начала работы над новой программой Delphi создает минимально необходимый код, обеспечивающий ее нормальное функционирование в Windows. Таким образом, простейшая программа готова сразу после выбора команды File ► New ► Application, и нам остается просто запустить программу. Однако до этого советую выполнить две важные вещи: создать собственный рабочий каталог (папку) и нужным образом настроить Delphi.

Настройка Delphi

В процессе работы над проектами программ, описываемых в этой книге, вам понадобится создавать множество форм и модулей. Полезно сохранять эти данные

в виде файлов в отдельной папке на диске. Более того, для каждой программы в этой папке имеет смысл создать свою вложенную папку. Тогда, чтобы освободить место на диске для серьезной программы, вам будет достаточно уничтожить основную папку, а чтобы передать ту или иную учебную программу своему коллеге, — переписать на дискету содержимое соответствующей вложенной папки. Создайте папку с именем, например, MY_DELPHI, а в ней — вложенную папку для вашей первой программы. Чтобы не утруждать себя придумыванием подходящих имен для «безликих» учебных программ, советую использовать для папки имя PGM_1 или что-нибудь подобное.

Второе, что нам предстоит сделать, — это внести изменения в стандартную настройку среды Delphi. Это необходимо для того, чтобы среда автоматически сохраняла на диске последнюю версию создаваемой вами программы. Выберите в главном меню команду Tools ► Environment Options и убедитесь, что в появившемся диалоговом окне активна вкладка Preferences. В левом верхнем углу этой вкладки в группе Autosave Options есть флажки Editor Files и Desktop. Установка флажков приведет к автоматическому сохранению текста окна кода программы и общего расположения окон Delphi перед началом очередного прогона создаваемой программы, что избавит вас от возможных потерь в случае ее «зависания». Советую установить эти флажки, щелкнув на каждом мышью. Чтобы следить за ходом компиляции, установите также флажок Show Compiler Progress в группе Compiling and Running. Далее, мне кажется полезным сохранять в коде программы комментарии — текстовые фрагменты, которые не влияют на работу программы, но делают ее текст более понятным. Если вы, как и я, предпочитаете писать комментарии на русском языке, следует выбрать соответствующий шрифт для отображения кода программы. По умолчанию редактор Delphi использует шрифт Courier New, в котором может не быть символов кириллицы (для версии Windows 95, например). В этом случае выберите команду Tools ► Editor Options, в открывшемся диалоговом окне перейдите на вкладку Display и в списке Editor Font выберите пункт Courier New Cyr.

Теперь все готово для прогона вашей первой программы. Щелкните мышью на кнопке  в главном окне или, что проще, нажмите клавишу F9: именно таким способом подготовленная в Delphi программа последовательно проходит три главных этапа своего жизненного цикла — этапы компиляции, компоновки и исполнения. На этапе компиляции осуществляется преобразование подготовленного в окне кода текста программы на языке Delphi в последовательность машинных инструкций, на этапе компоновки к ней подключаются необходимые вспомогательные подпрограммы, а на этапе исполнения готовая программа загружается в оперативную память и ей передается управление.

Как только вы нажмете клавишу F9, появится диалоговое окно Save Unit1 As, в котором Delphi попросит вас указать имя файла для модуля Unit1.pas и папку его размещения.

Лучше не идти «на поводу» у Delphi и не соглашаться с предложенным именем UNIT1.PAS, а присвоить модулю какое-нибудь более запоминающееся имя. Например, назовем модуль MY_1ST_U, что, по мысли автора, должно расшифровываться как «мой первый модуль» (разумеется, вам в голову может придти более подходящее имя). Если в окне Save Unit1 As вы укажете такое имя и нажмете клавишу Enter, Delphi попросит задать имя еще и для проекта программы в целом. Под этим име-

нем будет создан исполняемый (EXE) файл и здесь нет предела для вашей изобретательности, например, FIRSTPGM — вполне достойное имя для вашей первой программы в Delphi.

Имена в Delphi

Поскольку речь зашла об именах, я должен объяснить свою странную приверженность к сокращенным англоязычным именам. Дело в том, что Delphi принципиально не признает ни русских, ни французских, ни китайских, ни вообще каких бы то ни было имен, в которых используются символы, отличные от *латинских букв, цифр и знака подчеркивания*, причем имя не должно начинаться цифрой, но может начинаться знаком подчеркивания. Так как в этом перечне нет пробела, имена не могут также состоять из нескольких слов. Далее, сокращения до восьми символов — это дань ограничениям Delphi версии 1. Эта версия рассчитана на работу под управлением операционной системы Windows 3.x, которая не признает длинных имен файлов. Если вы работаете с 32-разрядной версией Delphi (то есть с любой из версий Delphi от 2 до 7), можете не сокращать имена и написать MY_FIRST_UNIT или даже MOY_PERVIJ_MODUL, потому что эти версии рассчитаны на современные 32-разрядные операционные системы, разрешающие использовать длинные имена файлов.

Если вы тем или иным способом решили проблемы с именами модуля и программы, на экране появится окно вашей формы, как и следовало ожидать, пустое, но обладающее всеми атрибутами нормального окна Windows-программы: оно может изменять свое положение и размеры и, закрываясь, выгружает программу из оперативной памяти.

Изменение свойств формы

Итак, первый и самый простой шаг при программировании в Delphi вами уже сделан: вами создана нормальная Windows-программа. Разумеется, эта программа не реализует придуманного вами алгоритма, она вообще ничего путного не умеет делать, но это — не беда! Впереди много времени и эта толстая книга, в которой как раз и рассказывается о том, как заставить вашу программу делать что-нибудь стоящее.

Теперь попробуем разобраться с тем, что такое модуль. В первом приближении мы можем считать модулем самостоятельный раздел программы, в чем-то подобный главе в книге. Модуль создается каждый раз, когда вы создаете новую форму (в программе может быть и часто бывает не одна, а несколько — иногда несколько десятков — форм и связанных с ними модулей). При компиляции программы Delphi создает файлы с расширениями PAS, DFM и DCU¹ для каждого модуля: PAS-файл содержит копию текста из окна кода программы, в файле с расширением DFM хранится описание содержимого окна формы, а в DCU-файле — результат преобразования в машинные инструкции текста из обоих файлов. DCU-файлы создаются компилятором и дают необходимую базу для работы компоновщика, который преобразует их в единый загружаемый файл с расширением EXE.

¹ Delphi 6 и 7 создает еще DPP-файл, в котором хранится информация о связанных с модулем диаграммах.

Попробуем модифицировать программу, например, изменим заголовок ее окна. По умолчанию (то есть без каких-либо усилий с нашей стороны) заголовок окна совпадает с заголовком формы — Form1. Чтобы изменить заголовок, нужно обратиться к окну инспектора объектов. Закройте окно работающей программы Form1 (если вы еще не сделали этого) и щелкните мышью в строке Caption (Заголовок) окна инспектора объектов. Теперь правая колонка этой строки с текстом Form1 выделена цветом и справа от выделенного виден текстовый мигающий курсор. Переключите клавиатуру в режим ввода кириллицы и введите заголовок **Моя первая программа**, после чего еще раз нажмите клавишу F9. При новом прогоне программы создается окно с заголовком **Моя первая программа**, что несет в себе гораздо больше информации, чем просто Form1.

Итак, простым изменением содержимого строки в окне инспектора объектов мы добились важной перемены: изменили одно из свойств окна программы — его заголовок. Таким же образом можно изменять любое другое свойство формы.

Размещение нового компонента

Разместить на форме новый компонент нетрудно. Для этого сначала нужно его выбрать (щелкнуть на нем мышью) в палитре компонентов, а затем щелкнуть мышью на точке рабочего пространства формы, где должен располагаться левый верхний угол компонента.

Попробуем таким способом включить в окно программы компонент Label (Метка), предназначенный для размещения различного рода надписей. Убедитесь в том,

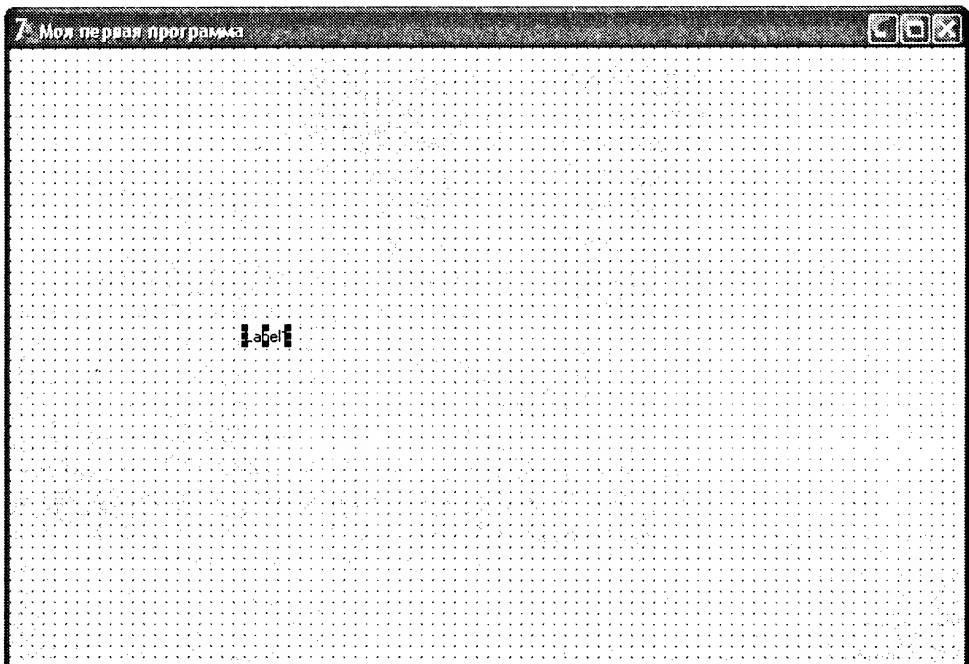


Рис. 2.1. Размещение компонента Label

что в палитре компонентов выбрана вкладка *Standard*, и щелкните мышью на кнопке **A** (эта кнопка отображает компонент *Label* в палитре компонентов). Теперь щелкните мышью на форме так, чтобы компонент появился на форме и располагался левее и выше ее центра (рис. 2.1). Первоначальные размеры и положение компонента на форме легко изменяются мышью, поэтому добиваться полного сходства с рисунком не обязательно.

Новый компонент имеет стандартное имя *Label1* и надпись на нем повторяет это имя. Изменим эту надпись: с помощью строки *Caption* окна инспектора объектов введите надпись *Я программирую на Delphi*. Как только вы начнете вводить новую надпись, вид компонента на форме начнет меняться, динамически отражая все изменения, производимые вами в окне инспектора объектов.

Выделим надпись цветом и сделаем ее шрифт более крупным. Для этого щелкните мышью на строке *Font* окна инспектора объектов и с помощью кнопки  в правой части строки раскройте диалоговое окно настройки шрифта. В списке *Size* (Размер) этого окна выберите высоту шрифта 24 пункта¹, а с помощью списка *Color* (Цвет) выберите нужный цвет (например, красный), после чего закройте окно щелчком на кнопке *OK*.

Надпись на компоненте в окне формы тут же соответствующим образом изменит свои свойства. *Delphi* обладает замечательной способностью визуальной реализации любых изменений свойств компонента не только на этапе прогона программы, но и на этапе проектирования формы.

Нажмите левую кнопку мыши внутри обрамляющих надпись черных прямоугольников и, не отпуская кнопку, сместите указатель мыши так, чтобы он расположился левее в центре окна, после чего отпустите кнопку. Таким способом можно буксировать компонент по форме, добиваясь нужного его положения.

С помощью обрамляющих черных квадратиков можно изменять размеры компонента. Для этого следует поместить указатель мыши на один из них (в этот момент указатель меняет свою форму на двунаправленную стрелку), затем нажать левую кнопку мыши и, не отпуская ее, буксировать сторону или угол компонента в нужном направлении, после чего отпустить кнопку.

Замечу, что все видимые компоненты² имеют свойства *Left* (Слева), *Top* (Сверху), *Width* (Ширина) и *Height* (Высота), числовые значения которых определяют положение левого верхнего угла компонента и его размеры в так называемых *пикселах*, то есть в минимальных по размеру точках экрана, светимостью которых может управлять программа. При буксировании компонента или изменении его размеров мышью эти значения автоматически меняются и наоборот — изменение этих свойств в окне инспектора объектов приводит к соответствующему изменению положения и размеров компонента. В *Delphi 7* значения *Left* и *Top* автоматически появляются во всплывающей подсказке (небольшом окне рядом с указателем мыши) при буксировке компонента по форме.

¹ Пункт — $1/72$ дюйма, то есть приблизительно 0,04 мм. Таким образом, 24 пункта означает высоту шрифта чуть более 9 мм.

² В *Delphi* могут использоваться как видимые, так и невидимые компоненты. Невидимые компоненты не имеют визуального отображения на этапе прогона программы.

Реакция на события

Настало время познакомиться с языком программирования Delphi, так как только с его помощью мы можем придать программе нужную функциональность и заставить ее выполнять полезную работу. Как уже говорилось, функциональность программы определяется совокупностью ее реакций на те или иные события. В связи с этим каждый компонент помимо свойств характеризуется также набором событий, на которые он может реагировать.

Модификация формы

Проведем очередную модернизацию нашей первой программы: вставим в ее форму еще один компонент — кнопку и заставим ее откликаться на событие, связанное со щелчком левой кнопки мыши.

Компонент *кнопка* изображается значком  на вкладке *Standard* палитры компонентов. Поместите этот компонент на форму и расположите его ниже метки и по середине формы (рис. 2.2).

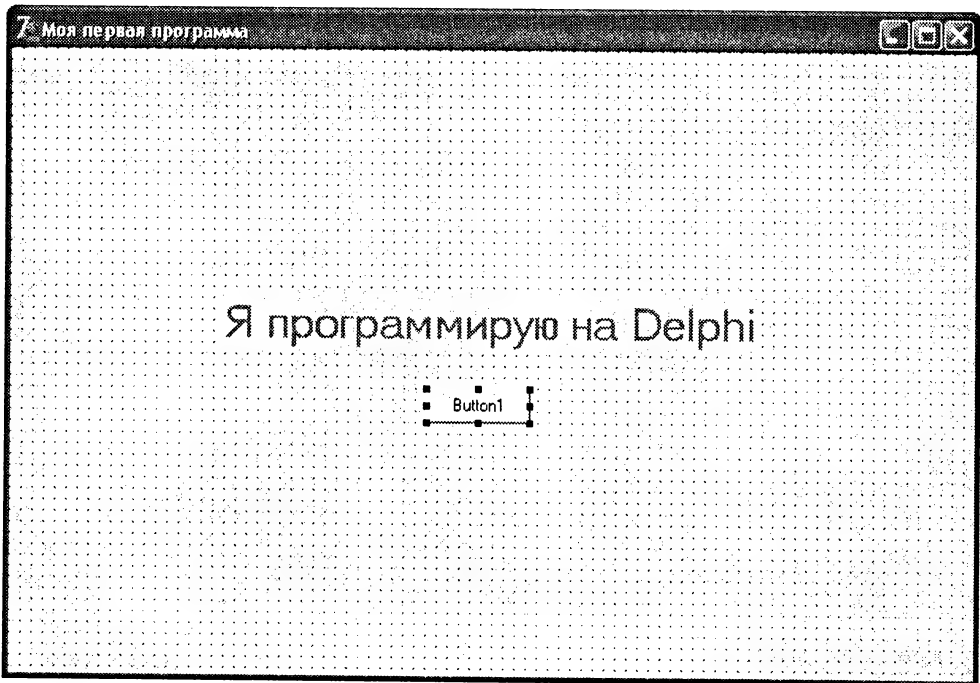


Рис. 2.2. Форма с вставленной кнопкой

Обработчик события OnClick

При щелчке на кнопке мышью в работающей программе возникает событие *OnClick* (По щелчку). Пока это событие никак не обрабатывается программой и поэтому щелчок на кнопке не приведет ни к каким последствиям. Чтобы заста-

вить программу реагировать на щелчок, необходимо написать на языке Delphi фрагмент программы, который называется *обработчиком события*. Этот фрагмент должен представлять собой последовательность текстовых строк, в которых программист указывает, что именно должна делать программа в ответ на щелчок на кнопке. Фрагмент оформляется в виде специальной подпрограммы языка Delphi — *процедуры*.

Чтобы заставить Delphi самостоятельно сделать заготовку для процедуры обработчика события `OnClick`, дважды подряд без заметной паузы щелкните мышью на вновь вставленном компоненте¹. В ответ Delphi активизирует окно кода, и вы увидите в нем такой текстовый фрагмент:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  
end;
```

Попробуем разобраться в том, что он содержит. Слово **procedure** извещает компилятор о начале подпрограммы-процедуры — в Delphi могут использоваться также подпрограммы-функции; в этом случае вместо слова **procedure** (процедура) используется слово **function** (функция); разницу между процедурами и функциями мы обсудим в главе 7. За ним следует имя процедуры `TForm1.Button1Click`. Это имя — составное: оно состоит из имени *класса* `TForm1` и собственно имени процедуры `Button1Click`.

Классами в Delphi называются функционально законченные фрагменты программ, служащие образцами для создания подобных себе экземпляров. Однажды создав класс, программист может включать его экземпляры (копии) в разные программы или в разные места одной и той же программы. Такой подход способствует максимально высокой продуктивности программирования за счет использования ранее написанных фрагментов программ. В состав Delphi входит несколько сотен классов, созданных программистами корпорации Borland (так называемых стандартных классов). Совокупность стандартных классов определяет мощные возможности этой системы программирования.

Каждый компонент принадлежит к строго определенному классу, а все конкретные экземпляры компонентов, вставляемые в форму, получают имя класса с добавленным числовым индексом. По используемому в Delphi соглашению все имена классов начинаются с буквы *T*. Таким образом, имя `TForm1` означает имя класса, созданного по образцу стандартного класса `TForm`. Если вы посмотрите начало текста в окне кода, то увидите следующие строки:

```
type  
TForm1 = class (TForm)  
  Button1: TButton;  
  Label1: TLabel;
```

¹ У начинающего пользователя Windows не всегда получается двойной щелчок. Этот нехитрый прием широко используется и в Windows, и в Delphi, так что вам придется потренироваться в его освоении: без него Delphi не вставит заготовку процедуры, и вам понадобится не только самостоятельно писать текст заготовки, но и производить дополнительные манипуляции с инспектором объектов и окном кода.

```
procedure Button1Click(Sender: TObject);  
private  
  { Private declarations }  
public  
  { Public declarations }  
end;
```

var

```
Form1: TForm1;
```

Представленная ниже строка определяет новый класс TForm1:

```
TForm1 = class(TForm)
```

Этот класс *порожден от* (создан по образцу) стандартного класса TForm. А следующая строка создает *экземпляр* этого класса с именем Form1:

```
Form1: TForm1;
```

Стандартный класс TForm описывает пустое окно программы, в то время как класс TForm1 описывает окно с уже вставленными в него компонентами *метка* и *кнопка*. Описание этих компонентов содержат строки:

```
Button1: TButton;  
Label1: TLabel;
```

Эти строки указывают, что компонент Button1 (Кнопка1) представляет собой экземпляр стандартного класса TButton, а компонент Label1 (Метка1) — экземпляр класса TLabel.

За именем процедуры TForm1.Button1Click в круглых скобках следует описание параметра вызова:

```
Sender: TObject
```

Это описание означает, что параметр с именем Sender принадлежит классу TObject. Как мы увидим дальше, процедуры могут иметь не один, а несколько параметров вызова или не иметь их вовсе. Параметры вызова (если они есть) служат для настройки реализованного в процедуре алгоритма на выполнение конкретной работы. Параметр Sender вставлен Delphi «на всякий случай»: с его помощью программист может, при желании, определить, какой именно компонент явился источником события OnClick.

Вся строка в целом называется *заголовком процедуры*:

```
procedure TForm1.Button1Click(Sender: TObject);
```

Ее завершает символ точки с запятой (;), который играет важную роль в Delphi, так как показывает компилятору на конец *предложения* языка. Из отдельных предложений составляется весь текст программы. В конце каждого предложения нужно ставить точку с запятой — это обязательное требование синтаксиса языка.

Три следующие строки определяют *тело процедуры*:

```
begin
```

```
end;
```

Слово **begin** (начало) сигнализирует компилятору о начале последовательности предложений, описывающих алгоритм работы процедуры, а слово **end** (конец) — о конце этой последовательности. В нашем случае тело процедуры пока еще не содержит описания каких-либо действий, что и неудивительно: среда Delphi лишь создала заготовку для процедуры, но она ничего «не знает» о том, для чего эта процедура предназначена. Наполнить тело нужными предложениями — задача программиста.

Для нас важно то обстоятельство, что каждый раз при щелчке на кнопке **Button1** управление будет передаваться в тело процедуры, а значит между словами **begin** и **end** мы можем написать фрагмент программы, который будет выполняться в ответ на это событие.

Чтобы убедиться в этом, сделаем нашу кнопку «звучащей»: напишите в пустой строке между словами **begin** и **end** следующее предложение:

```
MessageBeep (MB_OK) ;
```

Выполните прогон программы, предварительно включив звуковую систему компьютера. Теперь в ответ на щелчок на кнопке **Button1** в динамике компьютера будет раздаваться звуковой сигнал, так как вставленная строка реализует обращение к стандартной процедуре, которая умеет извлекать из динамика различные стандартные для Windows звуки.

Динамическое изменение свойств компонента

Поскольку кнопка **Button1** в нашей программе способна «звучать», полезно изменить ее надпись: вместо умалчиваемой надписи **Button1**, которую автоматически формирует Delphi по имени компонента, назовем кнопку, например, **Звук**. Проще всего это сделать с помощью окна формы и инспектора объектов, то есть на этапе конструирования формы (для этого нужно просто изменить свойство **Caption** компонента **Button1** в окне инспектора объектов), но для более полного знакомства с Delphi мы рассмотрим другой способ — способ динамического¹ изменения надписи на этапе прогона программы. Для этого создадим обработчик события **OnCreate** (По созданию) для формы и изменим в нем это свойство.

Событие **OnCreate** возникает после создания окна программы, но до появления этого окна на экране. Чтобы создать обработчик этого события, раскройте список компонентов в верхней части окна инспектора объектов, выберите компонент **Form1** и дважды щелкните на свойстве **OnCreate** вкладки **Events** этого компонента (щелкать нужно на правой части строки **OnCreate**). В ответ Delphi вновь активизирует окно кода и покажет вам заготовку для процедуры **TForm1.FormCreate**. Отредактируйте ее следующим образом:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
    Button1.Caption := 'Звук';  
end;
```

Единственная вставленная нами строка представляет собой так называемый *оператор присваивания* языка Delphi. В левой части оператора указывается свой-

¹ Изменения на этапе конструирования называются *статическими*, а в ходе прогона программы — *динамическими*.

ство `Button1.Caption`, а в правой части — значение Звук, которое мы хотим придать этому свойству. Связывает обе части комбинация символов «:=», которая читается как «присвоить значение». Символы «:=» всегда пишутся слитно, без разделяющих пробелов, хотя перед двоеточием и после знака равенства можно для удобочитаемости программы вставлять пробелы, что мы и сделали. Как и любое другое предложение языка, оператор присваивания завершается точкой с запятой.

Составное имя `Button1.Caption` необходимо для точного указания компилятору, о каком свойстве идет речь: в нашей программе используются три компонента (включая саму форму), каждый из которых имеет свойство `Caption`; уточняющий префикс `Button1` заставит изменить это свойство у кнопки, а не у метки или формы. Присваиваемое свойству значение является текстовой строкой. По правилам Delphi текстовая строка должна заключаться в обрамляющие апострофы. Внутри апострофов можно написать любое количество произвольных символов — именно они (без обрамляющих апострофов) будут определять новую надпись на кнопке.

После очередного прогона программы вы увидите измененную надпись на кнопке, а мы сделаем важный вывод: любое свойство любого компонента можно изменять *динамически*, то есть в ходе исполнения программы.

Некоторые итоги

Перед тем как навсегда расстаться с нашей первой программой, сформулируем главные выводы, полученные в ходе работы с ней.

- Процесс создания программы в Delphi разбивается на две фазы: фазу конструирования формы и фазу кодирования.
- Конструирование формы осуществляется путем выбора компонентов в палитре и размещения их на форме.
- Программист может перемещать любой размещенный на форме компонент и изменять его размеры с помощью мыши.
- Чтобы придать компоненту нужные свойства, используется вкладка `Properties` инспектора объектов.
- Чтобы компонент мог откликаться на то или иное событие, программист должен создать обработчик события и указать его имя на вкладке `Events` инспектора объектов.
- Обработчик события оформляется в виде процедуры, имеющей составное имя. Первая часть имени представляет собой имя класса для формы, вторая часть отделяется от первой точкой и может быть произвольной. Если Delphi автоматически формирует заготовку для обработчика, то вторая часть имени представляет собой объединение имени компонента и имени события без префикса `On`.
- Тело процедуры ограничено словами **begin** и **end** и состоит из отдельных предложений (операторов) языка Delphi. В конце каждого предложения ставится точка с запятой.
- Свойства компонента могут изменяться на этапе прогона программы.

Глава 3

Знакомство с компонентами

Как мы уже выяснили, компоненты представляют собой элементы, из которых конструируется видимое изображение, создаваемое работающей программой. Замечу, что существует значительное количество компонентов, которые не создают видимого изображения, но которые, тем не менее, играют важную роль в тех или иных случаях. Правильнее думать о компонентах как о заранее приготовленных для вас фрагментах программы, которые можно вставлять, если в этом есть необходимость, в разрабатываемую программу. В этой главе представлен начальный обзор компонентов, который даст вам самое общее представление о богатстве возможностей Delphi.

Библиотеки компонентов для разных версий Delphi строятся по принципу расширения: в первой версии было около 70 компонентов, в то время как в состав Delphi 7 входит более 300 компонентов.

ПРИМЕЧАНИЕ

Даже простое перечисление всех компонентов одной из последних версий заняло бы много места и вряд ли дало вам что-либо полезное. Поэтому ниже перечисляются лишь функциональные группы (вкладки палитры) компонентов и описываются те из них, что будут рассматриваться в рамках этой книги.

Компоненты перечисляются для комплектации Delphi 7 Studio Architect.

Вкладка Standard

На вкладке Standard палитры компонентов сосредоточены стандартные для Windows интерфейсные элементы, без которых не обходится практически ни одна программа.

- **Frame** — фрейм. Наравне с формой служит контейнером для размещения других компонентов. В отличие от формы, может размещаться в палитре компонентов, создавая заготовки компонентов.

- **MainMenu** — главное меню программы. Компонент способен создавать и обслуживать сложные иерархические меню.
- **PopupMenu** — контекстное, или локальное, меню. Обычно это меню появляется после щелчка правой кнопки мыши.
- **Label** — метка. Этот компонент используется для размещения в окне не очень длинных однострочных надписей.
- **Edit** — однострочное редактируемое текстовое поле. Предназначено для ввода, отображения или редактирования одной текстовой строки.
- **Мемо** — многострочное редактируемое текстовое поле. Используется для ввода и/или отображения многострочного текста.
- **Button** — кнопка. Обработчик события `OnClick` этого компонента обычно используется для реализации некоторой команды.
- **CheckBox** — флажок. Щелчок мышью на этом компоненте в работающей программе изменяет его логическое свойство `Checked`.
- **RadioButton** — переключатель. Обычно объединяется как минимум еще с одним таким же компонентом в группу. Щелчок на переключателе приводит к автоматическому освобождению ранее выбранного переключателя в той же группе.
- **ListBox** — список. Содержит список предлагаемых вариантов (пунктов списка) и дает возможность проконтролировать текущий выбор.
- **ComboBox** — комбинированный список. Представляет собой комбинацию списка и однострочного текстового поля.
- **ScrollBar** — полоса прокрутки. Представляет собой вертикальную или горизонтальную полосу, напоминающую полосы прокрутки по бокам окна программы.
- **GroupBox** — панель группировки. Этот компонент используется для группировки нескольких связанных по смыслу компонентов.
- **RadioGroup** — группа переключателей. Содержит специальные свойства для обслуживания нескольких связанных переключателей.
- **Panel** — панель. Этот компонент, как и `GroupBox`, служит для объединения нескольких компонентов. Содержит внутреннюю и внешнюю кромки, что позволяет создать эффекты «вдавленности» и «выпуклости».
- **ActionList** — список действий. Служит для централизованной реакции программы на действия пользователя, связанные с выбором одного из группы однотипных управляющих элементов, таких как команды меню, графические кнопки и т. п.

Вкладка Additional

На вкладку Additional помещены 26 дополнительных компонентов, с помощью которых можно разнообразить вид диалоговых окон.

- **BitBtn** — кнопка с изображением.
- **SpeedButton** — кнопка панели инструментов. Обычно используется для быстрого доступа к тем или иным командам главного меню.

- **MaskEdit** — поле с маской ввода. Этот компонент способен фильтровать вводимый текст, например, для правильного ввода даты.
- **StringGrid** — текстовая таблица. Этот компонент обладает мощными возможностями для представления текстовой информации в табличном виде.
- **DrawGrid** — произвольная таблица. В отличие от **StringGrid**, ячейки этого компонента могут содержать произвольную информацию, в том числе и рисунки.
- **Image** — изображение. Этот компонент предназначен для отображения рисунков, в том числе значков и метафайлов.
- **Shape** — стандартная фигура. С помощью этого компонента вы можете вставить в окно правильную геометрическую фигуру — прямоугольник, эллипс, окружность и т. п.
- **Bevel** — кромка. Служит для выделения отдельных частей окна трехмерными рамками или полосами.
- **ScrollBar** — панель с полосами прокрутки. В отличие от компонента **Panel**, автоматически вставляет полосы прокрутки, если размещенные в нем компоненты отсекаются его границами.
- **CheckBoxList** — список флажков. Отличается от стандартного компонента **ListBox** наличием рядом с каждым пунктом списка флажка **CheckBox**, что дает возможность выбора сразу нескольких пунктов.
- **Splitter** — вешка разбивки. Этот компонент размещается на форме между двумя другими видимыми компонентами и дает возможность пользователю во время прогона программы перемещать границу, отделяющую компоненты друг от друга.
- **StaticText** — текстовая метка. Отличается от стандартного компонента **Label** наличием собственного оконного ресурса, что позволяет обводить текст рамкой или выделять его в виде «вдавленной» части формы.
- **ControlBar** — контейнер для панелей инструментов. Служит контейнером для «причаливаемых» (**Drag&Dock**) компонентов.
- **ApplicationEvents** — обработчик сообщений **Windows**. Если этот компонент помещен на форму, он будет получать все предназначенные для программы сообщения **Windows** (без этого компонента сообщения принимает глобальный объект-программа **Application**).
- **ValueListEditor** — специализированный редактор списков, содержащих пары имя = значение. Пары такого типа широко используются в **Windows**, например, в файлах инициализации, в системном реестре и т. п.
- **LabeledEdit** — комбинация однострочного поля и метки.
- **ColorBox** — список выбора цвета. Специальный вариант компонента **ComboBox** для выбора одного из системных цветов.
- **Chart** — диаграмма. Этот компонент облегчает создание специальных панелей для графического представления данных.
- **ActionManager** — менеджер действий. Совместно с тремя следующими компонентами обеспечивает создание приложений, интерфейс которых (главное меню и инструментальные кнопки) может настраиваться пользователем.

- **ActionMainMenuBar** — строка меню для действий. Команды меню создаются с помощью компонента **ActionManager**.
- **ActionToolBar** — панель инструментов для действий. Служит контейнером для кнопок, создаваемых с помощью компонента **ActionManager**.
- **CustomizeDlg** — диалоговое окно настройки. С помощью этого компонента пользователь может по своему вкусу настроить интерфейс работающей программы.
- **XPColorMap** — совместно с тремя следующими компонентами впервые введен в версии 7 для настройки цветов и наполнения панелей **ActionToolBar**.
- **StandardColorMap** — в функциональном плане подобен компоненту **XPColorMap**.
- **WilightColorMap** — в функциональном плане подобен компоненту **XPColorMap**.

Вкладка Win32

Вкладка Win32 содержит интерфейсные элементы для 32-разрядных операционных систем Windows 95/98/NT/2000/XP.

- **TabControl** — набор вкладок. Каждая вкладка представляет собой прямоугольное поле с надписью и/или рисунком. Выбор той или иной вкладки распознается программой и используется для управления содержимым окна компонента.
- **PageControl** — набор страниц с вкладками. Каждая страница может содержать свой набор интерфейсных элементов и выбирается щелчком на связанной с ней вкладке.
- **ImageList** — хранилище изображений. Представляет собой хранилище для нескольких рисунков одинакового размера.
- **RichEdit** — поле формата RTF. В отличие от компонента **Memo** вкладки **Standard**, текст в компоненте **RichEdit** подчиняется правилам расширенного текстового формата (Rich Text Format, RTF) и может изменять такие свои характеристики, как шрифт, цвет, выравнивание и т. д.
- **TrackBar** — ползунок. Используется для управления значениями некоторых величин в программах. Например, с его помощью удобно изменять громкость звучания в мультимедийных программах.
- **ProgressBar** — индикатор процесса. С помощью этого компонента можно отображать ход исполнения достаточно длительного по времени процесса, например, процесса переноса данных на дискету.
- **UpDown** — счетчик. Две кнопки этого компонента предназначены для увеличения (верхняя) или уменьшения (нижняя) связанной с компонентом числовой величины.
- **HotKey** — клавиша быстрого выбора. Компонент используется для ввода управляющих клавиш, таких как F1, Alt+A, Ctrl+Shift+1 и т. п.
- **Animate** — анимация. Компонент предназначен для отображения последовательно сменяющих друг друга кадров движущихся изображений (видеоклипов). Компонент не может сопровождать видеоклип звуком.

- **DateTimePicker** — ввод и отображение времени/даты. Этот компонент предназначен для ввода и отображения даты или времени.
- **MonthCalendar** — календарь. Служит для выбора/отображения даты.
- **TreeView** — иерархическое дерево. Представляет собой совокупность связанных в древовидную структуру значков и строк. Обычно используется для просмотра структуры каталогов (папок) и других подобных элементов, связанных иерархическими отношениями.
- **ListView** — иерархический список. Организует просмотр нескольких значков и выбор нужного. Этот компонент способен располагать значки в горизонтальных или вертикальных рядах и показывать их в крупном или мелком масштабе.
- **HeaderControl** — управляющий заголовок. Представляет собой горизонтальную или вертикальную полосу, разделенную на ряд смежных секций с надписями. Размеры секций можно менять мышью на этапе работы программы. Обычно используется для изменения размеров столбцов или строк в разного рода таблицах.
- **StatusBar** — строка состояния. Предназначена для размещения разного рода служебной информации в окнах редактирования. Посмотрите на нижнюю часть окна кода Delphi или текстового редактора Word и вы увидите этот компонент в действии.
- **ToolBar** — инструментальная панель. Этот компонент служит контейнером для кнопок **ToolButton** и способен автоматически изменять их размеры и положение при удалении кнопок или при добавлении новых.
- **CoolBar** — инструментальная панель. В отличие от **ToolBar**, используется как контейнер для размещения стандартных интерфейсных компонентов Windows, таких как **Edit**, **ListBox**, **ComdoBox** и т. д.
- **PageScroller** — панель с кнопками прокрутки. Служит для размещения узких инструментальных панелей. При необходимости автоматически создает по краям панели стрелки прокрутки.
- **ComboBoxEx** — компонент в функциональном отношении подобен **ComboBox** (вкладка **Standard**), но имеет расширенные возможности.

Вкладка System

На вкладке **System** представлены компоненты, которые имеют различное функциональное назначение, в том числе компоненты, поддерживающие стандартные для Windows технологии межпрограммного обмена данными **OLE** (**Object Linking and Embedding** — связывание и внедрение объектов) и **DDE** (**Dynamic Data Exchange** — динамический обмен данными). Технология **DDE** в современных программах полностью вытеснена гораздо более мощной технологией **COM** и поэтому соответствующие им компоненты в книге не рассматриваются.

- **Timer** — таймер. Этот компонент служит для отсчета интервалов реального времени.
- **PaintBox** — окно для рисования. Создает прямоугольную область, предназначенную для прорисовки графических изображений.

- **MediaPlayer** — медиаплеер. С помощью этого компонента можно управлять различными мультимедийными устройствами.
- **OleContainer** — OLE-контейнер. Служит приемником связываемых или внедряемых объектов.

Вкладка Dialogs

Компоненты вкладки Dialogs реализуют стандартные для Windows диалоговые окна.

- **OpenDialog** — окно открытия файла. Реализует стандартное диалоговое окно открытия файла.
- **SaveDialog** — окно сохранения файла. Реализует стандартное диалоговое окно сохранения файла.
- **OpenPictureDialog** — окно открытия изображения. Реализует специальное окно выбора графических файлов с возможностью предварительного просмотра изображений.
- **SavePictureDialog** — окно сохранения изображений. Реализует специальное окно сохранения графических файлов с возможностью предварительного просмотра рисунков.
- **FontDialog** — окно выбора шрифта. Реализует стандартное диалоговое окно выбора шрифта.
- **ColorDialog** — окно выбора цвета. Реализует стандартное диалоговое окно выбора цвета.
- **PrintDialog** — окно настройки параметров печати. Реализует стандартное диалоговое окно выбора параметров для печати документа.
- **PrinterSetupDialog** — окно настройки параметров принтера. Реализует стандартное диалоговое окно для настройки печатающего устройства.
- **FindDialog** — окно поиска. Реализует стандартное диалоговое окно поиска текстового фрагмента.
- **ReplaceDialog** — окно поиска и замены. Реализует стандартное диалоговое окно поиска и замены текстового фрагмента.

Вкладка Win31

Большинство компонентов вкладки Win31 введены для совместимости с версией 1 Delphi. В современных программах вместо них рекомендуется использовать соответствующие компоненты вкладки Win32.

- **TabSet** — набор вкладок. В приложениях для 32-разрядных версий Windows рекомендуется использовать компонент **TabControl**.
- **OutLine** — иерархическая структура. В приложениях для 32-разрядных версий Windows рекомендуется использовать компонент **TreeView**.
- **TabbedNotebook** — набор страниц с вкладками. В приложениях для 32-разрядных версий Windows рекомендуется использовать компонент **PageControl**.

- **Notebook** — набор страниц без вкладок. В приложениях для 32-разрядных версий Windows рекомендуется использовать компонент **PageControl**.
- **Header** — управляющий заголовок. В приложениях для 32-разрядных версий Windows рекомендуется использовать компонент **HeaderControl**.
- **FileListBox** — панель с именами файлов.
- **DirectoryListBox** — панель с именами каталогов (папок).
- **DriveComboBox** — список устройств.
- **FilterComboBox** — список с расширениями файлов.

Вкладка Samples

Вкладка **Samples** содержит компоненты разного назначения.

- **Gauge** — индикатор величины. Подобен компоненту **ProgressBar** (вкладка **Win32**), но отличается большим разнообразием форм.
- **ColorGrid** — таблица цветов. Этот компонент предназначен для выбора основного и фоновых цветов в 16-цветной палитре.
- **SpinButton** — спаренные кнопки. Дает удобное средство управления некоторой числовой величиной.
- **SpinEdit** — поле со спаренными кнопками. Обеспечивает отображение и редактирование целого числа с возможностью его изменения с помощью спаренных кнопок.
- **DirectoryOutline** — структура каталогов. Отображает в иерархическом виде структуру каталогов дискового накопителя.
- **Calendar** — календарь. Предназначен для показа и выбора дня в месяце.

Вкладка ActiveX

Компоненты **ActiveX** являются «чужими» для Delphi: они создаются другими инструментальными средствами разработки программ (например, C++ или Visual Basic) и внедряются в Delphi с помощью технологии OLE. На вкладке **ActiveX** представлены лишь 4 из великого множества компонентов **ActiveX**, разрабатываемых повсюду в мире компаниями-производителями программных средств и отдельными программистами.

- **Chartfx** — интерактивный график. Дает программисту удобное средство включения в программу интерактивных (диалоговых) графиков.
- **VSSpell** — система проверки правописания. Осуществляет орфографическую проверку правильности написания английских слов.
- **F1Book** — электронная таблица. Позволяет создавать и использовать рабочие книги электронных таблиц, подобно тому, как это делает MS Excel.
- **VtChart** — мастер диаграмм. Обеспечивает мощные средства построения двух- и трехмерных диаграмм по результатам табличных вычислений.

Глава 4

Введение в язык Delphi

Основательному изучению языка Delphi посвящена часть II этой книги. Назначение данной главы — предоставить вам возможность неформального знакомства с основными свойствами языка программирования без точного описания и углубленного изучения его конструкций.

Учебная программа

При описании свойств языка мне понадобится иллюстрировать их небольшими примерами. В таких примерах чаще всего реализуются простые действия по вводу некоторых данных, их преобразованию и выводу результатов на экран. Для реализации этих действий создадим форму, которая будет содержать все необходимые для учебных примеров интерфейсные элементы. Чтобы эта форма была доступна в любой учебной программе, мы поместим ее в так называемое *хранилище объектов Delphi* — архив, в котором сохраняются заготовки для новых программ.

Вид формы для учебной программы показан на рис. 4.1.

На форме размещена панель (Panel), две кнопки (BitBtn), однострочное поле (Edit), метка (Label) и многострочное поле (Memo). Панель понадобилась для того, чтобы размещенные на ней кнопки, поле ввода и метка всегда располагались в нижней части окна, даже если окно развернуто на весь экран. Левая кнопка (OK) служит для сообщения программе о том, что в поле ввода подготовлен текст. Обработчик события OnClick этой кнопки станет тем самым местом программы, куда мы будем помещать учебные фрагменты. Правая кнопка (Close) служит для завершения работы программы. Ее функции аналогичны функциям системной кнопки закрытия окна и поэтому ее можно было бы и не размещать на форме. Однако я предпочитаю использовать крупную, бросающуюся в глаза кнопку, чтобы даже неопытный пользователь сразу увидел интерфейсный элемент, с помощью которого он сможет прекратить работу с программой. Метка располагается над полем ввода и пока не видна на экране. Она предназначена для размещения коротких сообщений, которые будет формировать программа. Для вывода многострочных сообщений служит многострочное поле Мемо, занимающее все остальное место окна.

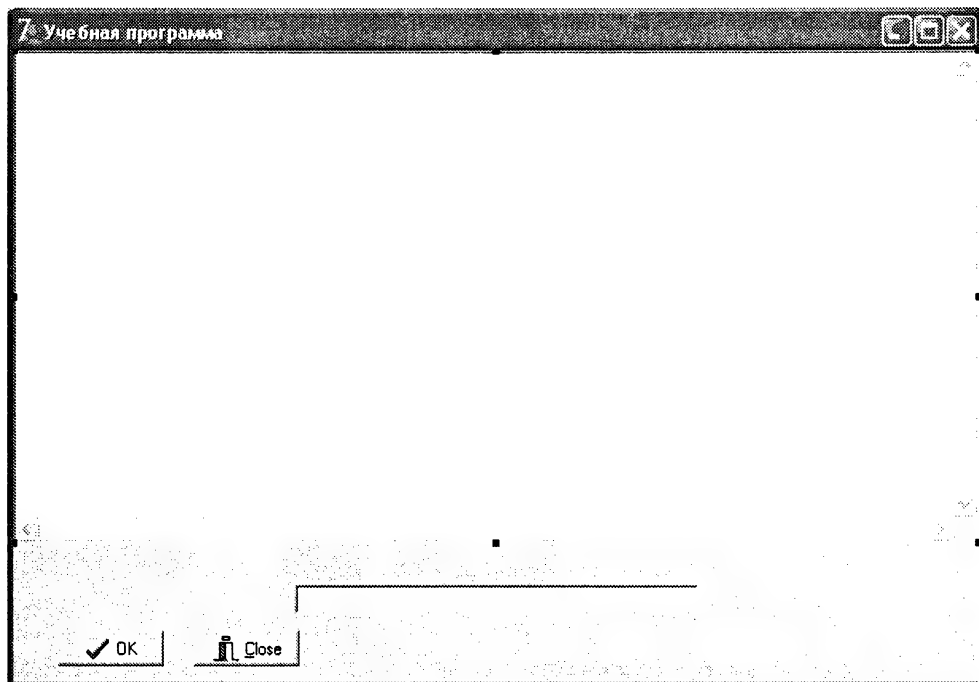


Рис. 4.1. Форма для учебной программы

Конструирование формы

Вы можете самостоятельно сконструировать форму для учебной программы, руководствуясь приведенным выше описанием размещенных на ней компонентов и рис. 4.1. Для облегчения этого процесса ниже для каждого компонента указаны свойства, значения которых отличаются от умалчиваемых. Компоненты перечислены в порядке их размещения, то есть вначале подготавливаются свойства формы, потом на форму помещается панель, на нее помещаются кнопки и строка ввода, а метка и многострочное текстовое поле — на свободное место формы над панелью.

Размеры компонентов установлены с учетом разрешения экрана 1024×768. Для другого разрешения эти размеры, возможно, потребуются скорректировать.

Форма.

- Caption — Учебная программа.
- Height — 480.
- Name — fmExample.
- Position — poScreenCenter.
- Width — 696.

Панель (компонент Panel на вкладке Standard) помещается на любое свободное место формы.

- Align — alBottom.
- BevelOuter — bvNone.
- Caption — очистить.
- Height — 84.

Кнопка OK (компонент BitBtn на вкладке Additional) помещается на любое свободное место панели.

- Kind — bkOk.
- Left — 32.
- Name — bbRun.
- Top — 48.

Кнопка Close (компонент BitBtn на вкладке Additional) помещается на любое свободное место панели.

- Kind — bkClose.
- Left — 128.
- Name — bbClose.
- Top — 48.

Поле ввода (компонент Edit на вкладке Standard) помещается на любое свободное место панели.

- Name — edInput.
- Left — 200.
- Text — очистить.
- Top — 16.
- Width — 289.

Метка (компонент Label на вкладке Standard) помещается на любое свободное место формы.

- Align — alBottom.
- Caption — очистить.
- Name — lbOutput.

Многострочное поле (компонент Memo на вкладке Standard) помещается на любое свободное место формы.

- Align — alClient.
- Lines — очистить.
- Name — mmOutput.
- ScrollBars — ssBoth.
- WordWrap — False.

Следует прокомментировать сделанные нами изменения значений свойств.

Размеры формы Height (Высота) и Width (Ширина) выбраны так, чтобы форма не казалась слишком маленькой или слишком большой на экранах с разным раз-

решением — они оптимальны для распространенного на сегодня разрешения 1024×768. Свойство `Position` (Положение) управляет размещением формы относительно границ экрана. Значение по умолчанию `poDesigned` требует разместить форму так, как она была размещена на этапе конструирования, рекомендуемое значение `poScreenCenter` — симметрично по центру экрана.

Свойство `Align` (Выравнивание) указывает способ выравнивания компонента относительно того контейнера, в котором он размещается. Контейнером называется компонент, предназначенный для размещения в нем других компонентов. Пустая форма представляет собой контейнер для других компонентов. Таким же контейнером является компонент `Panel`. Мы разместили на форме панель и для ее свойства `Align` установили значение `alBottom` (Внизу). После установки этого значения компонент (в данном случае — панель) прижимается к нижней границе формы и растягивается по всей ее длине. Теперь панель и все размещенные на ней компоненты будут примыкать к нижнему краю окна, даже если окно разворачивается на весь экран. Если бы мы разместили поле ввода и кнопки не на панели, а непосредственно на форме, они не имели бы свойства перемещения и оставались бы на своих местах при изменении размеров окна¹. В результате значительная часть полностью развернутого окна осталась бы пустой и не использовалась.

Свойство `Caption` (Заголовок) присуще многим видимым элементам. Для панели оно определяет текст, который размещается в ее центре. Поскольку мы не намерены размещать на панели никаких текстовых сообщений, это свойство у нее следует очистить.

Чтобы панель и расположенная над ней метка воспринимались как единое целое, в свойстве `BevelOuter` (Внешняя кромка) панели установлено значение `bvNone` (нет кромки).

Свойства `Left` (Слева), `Top` (Сверху) и `Width` (Ширина) определяют положение левого верхнего угла компонента относительно такого же угла своего контейнера и его (компонента) ширину. Указанные в списке значения этих свойств для поля ввода обеспечат симметричное по горизонтали расположение поля относительно сторон восстановленного (неразвернутого) окна (если, разумеется, вы установили рекомендуемую ширину формы 696). К сожалению, симметрия будет нарушена, если окно развернется. В Delphi есть средства изменения значений указанных свойств непосредственно в работающей программе. Создав обработчик события `OnResize` (По изменению размера) для формы, мы при желании могли бы научить программу реагировать на изменение размеров окна и соответствующим образом изменять значение свойства `Left` для строки ввода.

Свойство `Name` (Имя) определяет имя, под которым компонент будет известен программе. Я настоятельно рекомендую вам давать компонентам, на которые ссылается код программы, осмысленные имена вместо однотипных имен, которые «придумывает» Delphi. Более того, по принятому в современной практике программирования стилю именам обычно предшествует двухбуквенный префикс. Заложенная в префиксе мнемоника служит напоминанием программисту о типе поименованного компонента или его свойства. Например, все возможные значе-

¹ На самом деле это не совсем так. Компонент действительно не будет перемещаться при стандартных (умалчиваемых) значениях его свойства `Anchors`.

ния свойства `Align` имеют стандартный для Delphi префикс `al` (от `Align`), за которым следует вполне понятное английское слово: `alBottom`, `alTop`, `alLeft` и т. д. Поле ввода будет непременно участником кода большинства учебных программ, вот почему я предлагаю назвать этот компонент именем `edInput`: префикс `ed` — это сокращение английского названия типа компонента `Edit`, слово `Input` означает ввод данных в программу. Такими же осмысленными являются предлагаемые имена для метки (`lb` — от `Label`, `Output` — вывод программы), многострочного поля (`mm` — от `Memo`), кнопки `OK` (`bb` — от `BitBtn`, `Run` — пуск программы) и кнопки `Close`. Панель вряд ли будет упоминаться в программном коде, который вы будете писать, поэтому для нее можно оставить созданное Delphi умалчиваемое имя `Panel1`.

Свойство `Text` (Текст) у компонента `Edit` определяет текст, который будет содержать текстовое поле в момент появления на экране. Следует очистить это свойство, чтобы начальный текст не мешал вводу данных для программы. Точно так же следует очистить свойство `Caption` у метки и `Lines` (Строки) у многострочного поля. Чтобы очистить свойство `mmOutput.Lines`, щелкните в строке этого свойства в окне инспектора объектов и вызовите диалоговое окно изменения свойства. Затем удалите умалчиваемый текст `Memo1` (или `mmOutput`, если к этому моменту вы уже дали компоненту рекомендуемое имя) и закройте окно щелчком на кнопке `OK`.

Свойство `ScrollBars` (Полосы прокрутки) у компонента `mmOutput` имеет значение `ssBoth` (обе). Такое значение заставит Delphi поместить в компонент обе полосы прокрутки — вертикальную и горизонтальную. Наличие полос поможет вам увидеть текст, отсекаемый границами компонента. В то же время для свойства `WordWrap` (Перенос слов) советую установить значение `False`, чтобы отказаться от переноса слов на другую строку многострочного поля в длинных текстовых сообщениях. Для этого компонента свойство `Align` имеет значение `alClient`, что потребует от компонента занять всю оставшуюся незаполненной часть формы независимо от размеров окна.

Для компонентов `BitBtn` свойство `Kind` (Сорт) определяет типовые функции, которые реализует кнопка в диалоговых окнах. Значение `bkClose` определит типовой значок и надпись `Close`, которые Delphi разместит на кнопке, а также типовую функцию закрытия окна, которая будет связана с этой кнопкой. В результате без малейших усилий с нашей стороны мы получили кнопку, которая завершает работу программы. Значение `bkOK` заставит кнопку `BitBtn` закрывать диалоговое окно с результатом `mrOK`. Поскольку наша форма не является диалоговым окном, кнопка `OK` не будет выполнять эту свою функцию: мы использовали значение `bkOK` только для того, чтобы кнопка получила нужные значок и надпись.

Перенос формы в хранилище

После указанных выше изменений сохраните форму в файле с именем `Example` в каталоге хранилища Delphi. Этот каталог имеет имя `Objrepos` и размещается в каталоге размещения Delphi (по умолчанию — `c:\Program Files\Borland\Delphi7`). Хранилище Delphi служит для накопления типовых форм и проектов. Использование типовых заготовок стандартизует вид окон и способствует формированию собственного для программиста стиля оформления своих программных проектов.

Перенос формы в каталог `Objrepos` еще не обеспечивает включение этой формы в хранилище. Чтобы вызвать диалоговое окно регистрации формы в хранилище

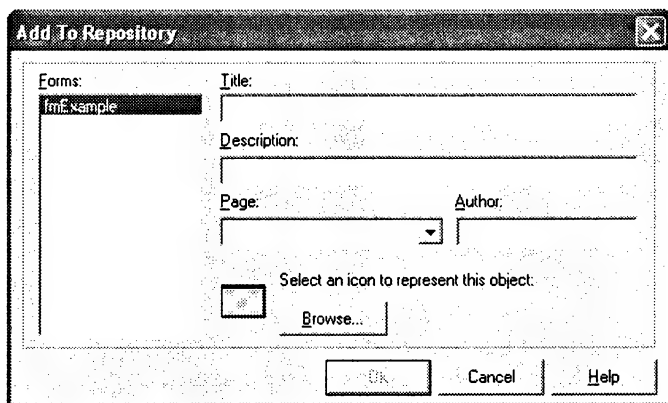


Рис. 4.2. Диалоговое окно регистрации формы в хранилище

(рис. 4.2), щелкните на форме правой кнопкой мыши и в появившемся контекстном меню выберите команду **Add To Repository**.

Не советую пользоваться командой **Project ► Add To Repository** главного меню, так как в этом случае в хранилище будет добавлен проект, а не форма. Проекты из хранилища копируются целиком вместе с многочисленными родительскими классами, что замедляет время создания нового проекта и приводит к неоправданным затратам дисковой памяти.

В поле **Title** введите имя формы **fmExample**, в поле **Description** — пояснение, например **Прототип главной формы для учебной программы**, раскройте список **Page** и выберите в нем пункт **Forms**. Если хотите, укажите в поле **Author** свою фамилию и закройте окно регистрации щелчком на кнопке **OK**. Теперь форма зарегистрирована, и вы в любой момент сможете выбрать ее с помощью команды **File ► New ► Other** главного меню Delphi. Однако я предлагаю не успокаиваться на этом и определить

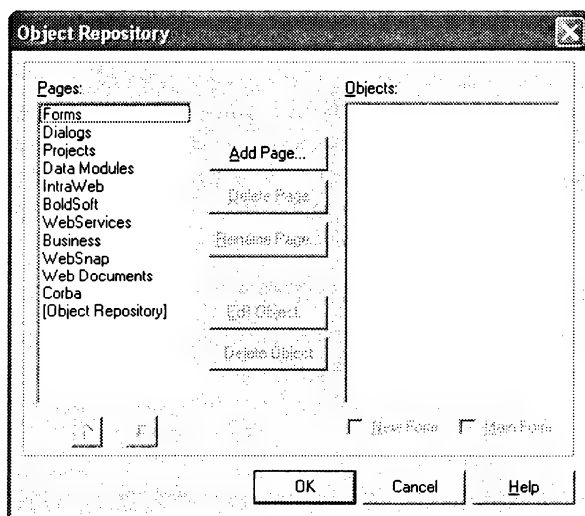


Рис. 4.3. Окно редактора хранилища

нашу форму как главную, тогда Delphi при выборе команды File ► New ► Application главного меню будет именно ее показывать вместо пустой типовой формы. Для этого с помощью команды Tools ► Repository вызовите диалоговое окно, показанное на рис. 4.3.

В списке Pages этого окна щелкните на пункте Forms и разыщите в самом конце списка объектов Objects нашу форму fmExample. Щелкните на ней, установите флажок Main Form (Главная форма) и закройте окно щелчком на кнопке OK. Когда нам в дальнейшем понадобится создавать проекты, основанные на стандартных пустых формах, вам потребуется вновь выбрать команду Tools ► Repository и в открывшемся окне редактора хранилища снять флажок Main Form.

Чтобы убедиться в правильности ваших действий, закройте проект, ответив щелчком на кнопке No в ответ на предложение Delphi сохранить файл проекта, и выберите команду File ► New ► Application — на экране должна появиться уже знакомая нам форма учебной программы.

Структура программ Delphi

Любая программа в Delphi состоит из файла проекта (файл с расширением dpr) и одного или нескольких модулей (файлы с расширениями pas). Каждый из таких файлов описывает *программную единицу* Delphi.

Структура проекта

Файл проекта представляет собой программу, написанную на языке Delphi и предназначенную для обработки компилятором. Эта программа автоматически создается Delphi и содержит лишь несколько строк. Чтобы увидеть их, запустите Delphi и выберите команду Project ► View Source в главном меню. Delphi покажет окно кода с вкладкой Project1, содержащей такой текст:

```
program Project1;

uses
    Forms,
    Unit1 in 'Unit1.pas' {fmExample};

{$R *.RES}

begin
    Application.Initialize;
    Application.CreateForm(TfmExample, fmExample);
    Application.Run;
end.
```

В окне кода жирным шрифтом выделяются так называемые зарезервированные слова, а курсивом — комментарии (также выделяются зарезервированные слова и комментарии в книге). Как видим, текст программы начинается зарезервированным словом **program** и заканчивается словом **end** с точкой за ним. Замечу, что сочетание **end** со следующей за ней точкой называется *терминатором программ*.

ной единицы: как только в тексте программы встретится такой терминатор, компилятор прекращает анализ программы и игнорирует оставшуюся часть текста.

Зарезервированные слова играют важную роль в Delphi, придавая программе в целом свойство текста, написанного на почти естественном английском языке. Каждое зарезервированное слово (а их в Delphi несколько десятков) несет в себе условное сообщение для компилятора, который анализирует текст программы так же, как читаем его и мы: слева направо и сверху вниз.

Комментарии, наоборот, ничего не значат для компилятора, и он их игнорирует. Комментарии важны для программиста, который с их помощью поясняет те или иные места программы. Наличие комментариев в тексте программы делает ее понятнее и позволяет легко вспомнить особенности реализации программы, которую вы написали несколько лет назад. В Delphi комментарием считается любая последовательность символов, заключенная в фигурные скобки. В приведенном выше тексте таких комментариев два, но следующая строка на самом деле не является комментарием:

```
{ $R *.RES }
```

Этот специальным образом написанный фрагмент кода называется *директивой компилятора* (в нашем случае — это указание компилятору на необходимость подключения к программе так называемого файла ресурсов). Директивы начинаются символом \$, который стоит сразу за открывающей фигурной скобкой.

Поскольку речь зашла о комментариях, замечу, что в Delphi в качестве ограничителей комментария могут также использоваться пары символов (*, *) и //. Скобки (*...*) используются подобно фигурным скобкам, то есть комментарием считается находящийся в них фрагмент текста, а символы // указывают компилятору, что комментарий располагается за ними и продолжается до конца текущей строки:

```
{ Это комментарий }
(* Это тоже комментарий *)
// Все символы до конца этой строки составляют комментарий
```

Слово **program** со следующим за ним именем программы и точкой с запятой образуют *заголовок программы*. За заголовком следует раздел описаний, в котором программист (или Delphi) описывает используемые в программе идентификаторы. Идентификаторы обозначают элементы программы, такие как типы, переменные, процедуры, функции (подробнее об элементах программы мы поговорим чуть позже). Здесь же с помощью предложения, которое начинается зарезервированным словом **uses** (используется) программист сообщает компилятору о тех фрагментах программы (модулях), которые необходимо рассматривать как неотъемлемые составные части программы и которые располагаются в других файлах. Следующие строки указывают, что помимо файла проекта в программе должны использоваться модули Forms и Unit1:

```
uses
  Forms,
  Unit1 in 'Unit1.pas' {fmExample};
```

Модуль Forms является стандартным (то есть уже известным Delphi), а модуль Unit1 — новым, ранее неизвестным, и Delphi в этом случае указывает также

имя файла с текстом модуля (**in** 'Unit1.pas') и имя связанного с модулем файла описания формы {fmExample}.

Собственно тело программы начинается со слова **begin** (начать) и ограничивается терминатором **end** с точкой. Тело состоит из нескольких операторов языка Delphi. В каждом операторе реализуется некоторое действие — изменение значения переменной, анализ результата вычисления, обращение к подпрограмме и т. п. В теле нашей программы — три исполняемых оператора:

```
Application.Initialize;
Application.CreateForm(TfmExample, fmExample);
Application.Run;
```

Каждый из них реализует обращение к одному из *методов объекта* Application. Я вынужден забежать вперед и хотя бы кратко пояснить только что введенные термины, которые определяют важные для Delphi понятия. *Объектом* называется специальным образом оформленный фрагмент программы, заключающий в себе данные и подпрограммы для их обработки. Данные называются *полями* объекта, а подпрограммы — его *методами*. Объект в целом предназначен для решения какой-либо конкретной задачи и воспринимается в программе как неделимое целое (иными словами, нельзя из объекта «выдернуть» отдельное поле или метод). Объекты играют чрезвычайно важную роль в современных языках программирования. Они придуманы для того, чтобы увеличить производительность труда программиста и одновременно повысить качество разрабатываемых им программ. Два главных свойства объекта — функциональность и неделимость — делают его самостоятельной или даже самодостаточной частью программы и позволяют легко переносить объект из одной программы в другую. Разработчики Delphi придумали для нас с вами сотни объектов, которые можно рассматривать как кирпичики, из которых программист строит многоэтажное здание программы. Такой принцип построения программ называется объектно-ориентированным программированием (ООП).

В объекте Application собраны данные и подпрограммы, необходимые для нормального функционирования Windows-программы в целом. Delphi автоматически создает объект-программу Application для каждого нового проекта. Следующая строка означает обращение к методу Initialize объекта Application:

```
Application.Initialize;
```

Прочитав эту строку, компилятор создаст код, который заставит процессор перейти к выполнению некоторого фрагмента программы, написанного для нас разработчиками Delphi. После выполнения этого фрагмента (программисты говорят: после *выхода* из подпрограммы) управление процессором перейдет к следующей строке программы, в которой вызывается метод CreateForm и т. д.

ПРИМЕЧАНИЕ

Метод Application.Initialize вызывается «на всякий случай»: в реальной программе он обычно ничего не делает. Его назначение — выполнить подпрограмму, имя которой размещается в системной переменной InitProc. В некоторых специальных случаях один из модулей программы в своей иницилирующей секции может присвоить переменной имя программы, которая должна выполняться сразу после загрузки всех модулей, но перед началом работы основной программы.

Метод `CreateForm` объекта `Application` создает и показывает на экране окно главной формы, а метод `Run` реализует бесконечный цикл получения и обработки поступающих от Windows сообщений о действиях пользователя. Когда пользователь щелкнет на кнопке `Close`, Windows передаст программе специальное сообщение, которое, в конечном счете, заставит программу прекратить работу и освободить назначенные ей системные ресурсы (память и процессор).

Следует сказать, что файл проекта полностью формируется самой средой Delphi и в большинстве случаев не предназначен для редактирования. Именно по этой причине файл проекта имеет особое расширение и обычно не виден в окне кода. При разработке сложных проектов этот файл можно использовать для выполнения некоторых специфических действий перед началом работы программы.

Структура модуля

Модули — это программные единицы, предназначенные для размещения отдельных частей программ. С помощью содержащегося в них программного кода реализуется вся поведенческая сторона программы.

Любой модуль имеет следующую структуру¹:

- 1) заголовок;
- 2) секция интерфейсных объявлений;
- 3) секция реализаций;
- 4) терминатор.

Заголовок открывается зарезервированным словом **unit**, за которым следует имя модуля и точка с запятой. Секция интерфейсных объявлений открывается зарезервированным словом **interface**, а секция реализаций — словом **implementation**. Терминатором модуля, как и терминатором программы, является слово **end** с точкой. Следующий фрагмент программы является синтаксически правильным вариантом модуля:

```
unit Unit1;
interface
// Секция интерфейсных объявлений
implementation
// Секция реализаций
end.
```

В секции интерфейсных объявлений описываются программные элементы (типы, классы, процедуры и функции), которые будут «видны» другим программным модулям, а в секции реализаций раскрывается механизм работы этих элементов. Разделение модуля на две секции обеспечивает удобный механизм обмена алгоритмами между отдельными частями одной программы. Он также реализует средство обмена программными разработками между отдельными программистами. Получив откомпилированный «посторонний» модуль, программист получает доступ только к его интерфейсной части, в которой, как уже говорилось, содержатся объявления элементов. Детали реализации объявленных процедур, функций, классов скрыты в секции реализаций и недоступны другим модулям.

¹ Более точное описание структуры модуля приводится в главе 11.

Перейдите на вкладку Unit1 окна кода, и вы увидите такой текст:

```
unit Unit1;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls,  
  Forms, Dialogs, StdCtrls, Buttons, ExtCtrls;
```

```
type
```

```
  TfmExample = class (TForm)
```

```
    Panell: TPanel;
```

```
    bbRun: TBitBtn;
```

```
    bbClose: TBitBtn;
```

```
    edInput: TEdit;
```

```
    lbOutput: TLabel;
```

```
    mmOutput: TMemo;
```

```
  private
```

```
    { Private declarations }
```

```
  public
```

```
    { Public declarations }
```

```
  end;
```

```
var
```

```
  fmExample: TfmExample;
```

```
implementation
```

```
{ $R *.DFM }
```

```
end.
```

Весь этот текст сформирован Delphi, но, в отличие от файла проекта, программист может его изменять, придавая программе нужную функциональность.

В интерфейсной секции описан один тип (класс TfmExample) и один объект (переменная fmExample).

Вот описание класса:

```
type
```

```
  TfmExample = class (TForm)
```

```
    Panell: TPanel;
```

```
    bbRun: TBitBtn;
```

```
    bbClose: TBitBtn;
```

```
    edInput: TEdit;
```

```
    lbOutput: TLabel;
```

```
    mmOutput: TMemo;
```

```
  private
```

```

    { Private declarations }
public
    { Public declarations }
end;

```

Как мы увидим дальше, классы служат основным инструментом реализации мощных возможностей Delphi. Класс является образцом, по которому создаются объекты, и наоборот, объект — это экземпляр реализации класса. Образцы для создания элементов программы в Delphi называются *типами*, таким образом класс TfmExample1 — это тип. Перед его объявлением стоит зарезервированное слово **type** (тип), извещающее компилятор о начале раздела описания типов.

Стандартный класс TForm реализует все нужное для создания и функционирования пустого окна программы. Класс TfmExample1 *порожден* от этого класса, о чем свидетельствует строка:

```
TfmExample = class (TForm)
```

В этой строке за зарезервированным словом **class** в скобках указывается имя родительского класса. Термин «порожден» означает, что класс TfmExample унаследовал все возможности родительского класса TForm и добавил к ним собственные в виде дополнительных компонентов, которые, как вы помните, мы вставили в форму fmExample. Перечень вставленных нами компонентов и составляет значительную часть описания класса.

Свойство наследования классами-потомками всех свойств родительского класса и обогащения их новыми возможностями является одним из фундаментальных принципов объектно-ориентированного программирования. От наследника может быть порожден новый наследник, который внесет свою лепту в виде дополнительных программных заготовок и т. д. В результате создается ветвящаяся иерархия классов, на вершине которой располагается самый простой класс TObject (все остальные классы в Delphi порождены от этого единственного прародителя), а на самой нижней ступени иерархии — мощные классы-потомки, которым по плечу решение любых проблем.

Объект fmExample формально относится к элементам программы, которые называются *переменными*. Вот почему перед объявлением объекта стоит зарезервированное слово **var** (от английского *variables* — переменные).

СОВЕТ Текст модуля доступен как Delphi, так и программисту. Delphi автоматически вставляет в текст модуля описание любого добавленного к форме компонента, а также создаст заготовки для обработчиков событий; программист может добавлять свои методы в ранее объявленные классы, наполнять обработчики событий конкретным содержанием, вставлять собственные переменные, типы, константы и т. д. Совместное с Delphi владение текстом модуля окажется вполне успешным, если программист будет соблюдать простое правило: он не должен удалять или изменять строки, которые вставлены не им, а Delphi.

Элементы программы

Элементы программы — это минимальные неделимые ее части, имеющие определенное значение для компилятора. К элементам относятся:

- зарезервированные слова;
- идентификаторы;
- типы;
- константы;
- переменные;
- метки;
- подпрограммы;
- комментарии.

Зарезервированные слова — это английские слова, указывающие компилятору на необходимость выполнения определенных действий. Зарезервированные слова не могут использоваться в программе ни для каких иных целей кроме тех, для которых они предназначены. Например, зарезервированное слово **begin** означает для компилятора начало составного оператора. Программист не может создать в программе переменную с именем **begin**, константу **begin**, метку **begin** или вообще какой бы то ни было другой элемент программы с именем **begin**.

Идентификаторы — это слова, которыми программист обозначает любой другой элемент программы, кроме зарезервированного слова, идентификатора или комментария. Идентификаторы в Delphi могут состоять из латинских букв, арабских цифр и знака подчеркивания. Никакие другие символы или специальные знаки не могут входить в идентификатор. Из этого простого правила следует, что идентификаторы не могут состоять из нескольких слов (нельзя использовать пробел) или включать в себя символы кириллицы (русского алфавита).

Типы — это специальные конструкции языка, которые рассматриваются компилятором как образцы для создания других элементов программы, таких как переменные, константы и функции. Любой тип определяет две важные для компилятора вещи: объем памяти, выделяемый для размещения элемента (константы, переменной или возвращаемого функцией результата), и набор допустимых действий, которые программист может совершать над элементами данного типа. Замечу, что любой определяемый программистом идентификатор должен быть описан в разделе описаний (перед началом исполняемых операторов). Это означает, что компилятор должен знать тот тип (образец), по которому создается определяемый идентификатором элемент.

Константы определяют области памяти, которые не могут изменять своего значения в ходе работы программы. Как и любые другие элементы программы, константы могут иметь свои собственные имена. Объявлению имен констант должно предшествовать зарезервированное слово **const** (от английского *constants* — константы). Например, мы можем определить следующие константы:

```
const
  Kbyte = 1024;
  Mbyte = Kbyte*Kbyte;
  Gbyte = 1024*Mbyte;
```

Эти константы позволяют вместо длинных чисел 1048576 (1024×1024) и 1073741824 ($1024 \times 1024 \times 1024$) писать в тексте программы, соответственно, Mbyte

и `Gbyte`. Тип константы определяется способом ее записи и легко распознается компилятором в тексте программы, поэтому программист может не использовать именованные константы (то есть не объявлять их в программе явно).

Переменные связаны с изменяемыми областями памяти, то есть с такими ее участками, содержимое которых будет меняться в ходе работы программы. В отличие от констант, переменные всегда объявляются в программе. Для этого после идентификатора переменной ставится двоеточие и имя типа, по образу которого должна строиться переменная. Разделу объявления переменной (переменных) должно предшествовать слово **var**. Например:

```
var
  inValue: Integer;
  byValue: Byte;
```

Здесь идентификатор `inValue` объявляется как переменная типа `Integer`, а идентификатор `byValue` — как переменная типа `Byte`. Стандартный (то есть заранее определенный в Delphi) тип `Integer` задает четырехбайтовый участок памяти, содержимое которого рассматривается как целое число в диапазоне от -147 483 648 до +2 147 483 647, а стандартный тип `Byte` — участок памяти длиной 1 байт, в котором размещается беззнаковое целое число в диапазоне от 0 до 255.

Метки — это имена операторов программы. Метки используются очень редко и только для того, чтобы программист смог указать компилятору, какой оператор программы должен выполняться следующим. Метки, как и переменные, всегда объявляются в программе. Разделу объявлений меток предшествует зарезервированное слово **label** (метка). Например:

```
label
  Loop;
begin
  ...
  goto Loop; // Программист требует передать управление
              // оператору, помеченному меткой Loop.
  ...       // Эти операторы будут пропущены
  Loop:     // Оператору, идущему за этой меткой,
  ...       // будет передано управление
end;
```

Подпрограммы — это специальным образом оформленные фрагменты программы. Замечательной особенностью подпрограмм является их значительная независимость от остального текста программы. Говорят, что свойства подпрограммы *локализуются* в ее теле. Это означает, что если программист что-либо изменит в подпрограмме, ему, как правило, не понадобится в связи с этим изменять что-либо вне подпрограммы. Таким образом, подпрограммы являются средством структурирования программ, то есть расчленения программ на ряд во многом независимых фрагментов. Структурирование неизбежно для крупных программных проектов, поэтому подпрограммы используются в программах на Delphi очень часто.

В Delphi есть два сорта подпрограмм: процедуры и функции. Функция отличается от процедуры только тем, что ее идентификатор можно наряду с константами

и переменными использовать в выражениях, так как функция имеет выходной результат определенного типа. Пусть, например, определена функция:

```
function MyFunction: Integer;
```

Кроме того, пусть определена переменная:

var

```
X: Integer;
```

Тогда возможен такой оператор присваивания:

```
X := 2*MyFunction-1;
```

Имя процедуры нельзя использовать в выражении, так как процедура не имеет связанного с нею результата:

```
Procedure MyProcedure;
```

• • •

```
X := 2*MyProcedure-1; // Ошибка!
```

Типы

Типы в Delphi играют огромную роль. Связано это с тем, что лежащий в основе языка Delphi язык Pascal был специально придуман как средство обучения студентов программированию. Поскольку начинающий программист легко может допустить ошибку или неточно описать свои действия, компилятор Pascal должен был иметь средства контроля за действиями программиста, чтобы вовремя предостеречь его от последствий неверных действий. Первоначально типы как раз и предназначались для того, чтобы программист явно указывал компилятору, какого размера память нужна ему в программе и что он собирается делать с этой памятью. Практика применения типов показала их высокую эффективность для защиты программы от случайных ошибок, так что практически все современные языки программирования в той или иной степени реализуют механизм типов. В Delphi значение типов возросло в еще большей степени, так как с их помощью определяются классы — основной инструмент программиста.

Строковый и символьный типы

Свое знакомство с типами мы начнем со строкового типа **String**. Этот тип определяет участок памяти переменной длины, каждый байт которого содержит один символ. Для символов в Delphi используется тип Char, таким образом, тип **String** — это цепочка следующих друг за другом символов типа Char. Каждый символ в **String** пронумерован, причем первый символ имеет номер 1. Программист может обращаться к любому символу строки, указывая его порядковый номер в квадратных скобках сразу за именем переменной:

```
var           // Начало раздела описания переменных
S: String;   // Объявление переменной строкового типа
begin        // Начало раздела исполняемых операторов
S := 'Строка символов'; // Переменная S содержит
                        // значение "Строка символов"
```

```

S[6] := 'и';           // Теперь переменная содержит
                        // значение "Строка символов"
end;                  // Конец раздела исполняемых операторов

```

Наличие комментариев избавляет меня от необходимости подробного описания назначения каждой строки текста. Напомню, что программист обязан объявить любой вводимый им идентификатор. Идентификатор *S* неизвестен Delphi — он введен нами для переменной строкового типа. После его объявления в разделе переменных Delphi выделит для него начальную область памяти минимальной длины и будет контролировать использование этого идентификатора в разделе исполняемых операторов. Например, пусть в разделе исполняемых операторов встретилось следующее выражение:

```
2*S-1
```

Компилятор сразу забьет тревогу, так как строка не может быть участником математических вычислений: она предназначена для хранения символов, а не чисел.

В первом операторе присваивания в переменную *S* будет помещено значение строковой константы 'Строка символов'. Строковые константы содержат произвольные символы, заключенные в обрамляющие апострофы, причем сами апострофы не входят в значение константы, поэтому после присваивания переменная примет значение *Строка символов* без апострофов. Если понадобится включить в текстовую константу апостроф, он удваивается: 'Символ ' — это апостроф'. После первого присваивания *S* будет занимать участок памяти длиной 15 байт — по одному байту на каждый символ значения¹. Переменность размера области памяти, выделяемой для размещения строки символов, — характерная особенность типа **String**. Пусть, например, во втором операторе мы обратились бы не к 6-му по счету символу, а ко всей строке в целом:

```
S := 'и';
```

Тогда эта переменная стала бы занимать 1 байт, а следующие за ним 14 байтов оказались бы свободными. Длина строковой переменной в программе меняется автоматически при каждом присваивании переменной нового значения и может составлять от 0 до 2 Гбайт.

Над строковым типом определена операция сцепления (+):

```
S := 'Object'+' Pascal'; // S содержит "Object Pascal"
```

Кроме того, строки можно сравнивать с помощью таких операций отношения:

- = — равно;
- <> — не равно;
- > — больше;
- >= — больше или равно;
- < — меньше;
- <= — меньше или равно.

¹ На самом деле переменная строкового типа всегда имеет длину 4 байта, которые используются для указания на область памяти, в которой помещается сама строка символов. Подробнее о механизме использования памяти строковыми переменными см. главу 6.

ПРИМЕЧАНИЕ Символы <>, <= и >= пишутся слитно, их нельзя разделять пробелами или комментариями.

Результат применения операции сравнения к двум строкам имеет логический тип, который характеризуется двумя возможными значениями: True (Истина; читается *тру*) и False (Ложь; читается *фолс*). Строки сравниваются побайтно, слева направо; каждая пара символов сравнивается в соответствии с их внутренней кодировкой (см. главу 6).

Все остальные действия над строками осуществляются с помощью нескольких стандартных для Delphi подпрограмм, описанных в главе 6.

Создадим программу, в которой текст из однострочного поля `edInput` будет копироваться без изменений в метку `lbOutput` и многострочное поле `mmOutput`. Для этого загрузите Delphi и выберите команду **File ▶ New ▶ Application**.

СОВЕТ Как упоминалось в главе 2, имеет смысл для каждой учебной программы создавать свой вложенный каталог и размещать в нем все связанные с программой файлы. Если вы еще не сделали этого, сверните окно Delphi и создайте с помощью Проводника вложенный каталог `СОРУТХТ` в каталоге для ваших учебных программ. В дальнейшем я не буду напоминать вам о необходимости создания вложенных каталогов, однако помните, что если вы не будете этого делать, очень скоро ваш учебный каталог станет представлять собой хаотическое нагромождение никак не связанных друг с другом файлов.

Чтобы сообщить программе о том, что в поле `edInput` подготовлен текст, пользователь должен щелкнуть на кнопке `bbRun`, поэтому все необходимые действия мы сосредоточим в обработчике события `OnClick` этой кнопки. Дважды щелкните на кнопке `bbRun` в окне формы, чтобы среда Delphi автоматически подготовила для вас заголовок обработчика события, и отредактируйте его следующим образом:

```
procedure TfmExample.bbRunClick(Sender: TObject);
begin
    lbOutput.Caption := edInput.Text; // Повторяем текст в метке
    mmOutput.Lines.Add(edInput.Text); // и в многострочном поле
    edInput.Text := '';               // Очищаем однострочное поле
    edInput.SetFocus;                 // Передаем ему фокус ввода
end;
```

Нажмите клавишу F9 для прогона программы, сохраните модуль под именем `СоруТхтU.pas`, а проект в целом — под именем `СоруТхт.dpr` и вы увидите окно, показанное на рис. 4.4.

Вводимый текст хранится в свойстве `Text` компонента `edInput`. После выполнения первого оператора присваивания он будет перенесен в свойство `Caption` компонента `lbOutput` и тут же появится на экране над полем ввода.

Многострочное поле `mmOutput` способно сохранять и отображать на экране множество строк. Все они содержатся в его свойстве `Lines`, относящемся к классу `TStringList` (подробнее об этом классе см. главу 13). Метод `Add` класса `TStringList` добавляет новую строку к имеющемуся в свойстве `Lines` набору строк. Добавленная с помощью второго оператора строка тут же отображается на экране.

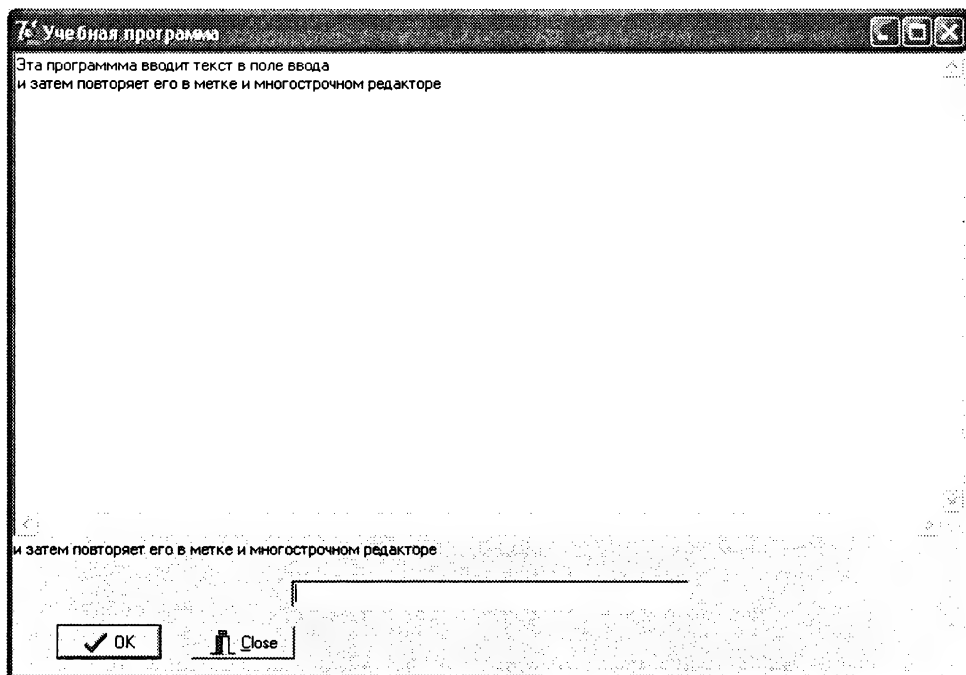


Рис. 4.4. Окно программы CopyTxt

С помощью следующего оператора строка ввода очищается и подготавливается к вводу нового текста:

```
edInput.Text := '';
```

Двойной апостроф в правой части оператора означает «пустую» строку или, что то же, строку нулевой длины.

Завершает обработчик оператор:

```
edInput.SetFocus;
```

С помощью этого оператора поле `edInput` получает фокус ввода. Термином «фокус ввода» в Delphi фактически обозначается клавиатура. «Получить фокус ввода» означает связать указанный компонент с клавиатурой. В окне программы может быть несколько компонентов, способных принимать и обрабатывать клавиатурный ввод (в нашей программе таких компонентов четыре: многострочное поле `mmOutput`, поле `edInput` и кнопки `bbRun` и `bbClose`). Каждый из таких компонентов имеет метод `SetFocus`, с помощью которого программа может передать компоненту контроль над клавиатурой. Чтобы оценить действия, реализуемые оператором `edInput.SetFocus`, поставьте в самом начале оператора признак комментария (символы `//`), сделайте новый прогон программы и введите с ее помощью хотя бы 2–3 строки.

Замечу, что в момент появления окна программы на экране поле `edInput` не имеет фокуса ввода, что затрудняет ввод первой строки. Попробуйте самостоятельно создать обработчик события `OnActivate` для формы `fmExample` и в нем установить фокус ввода для строки `edInput`.

СОВЕТ Раскройте список в верхней части окна инспектора объектов, выберите в нем компонент `fmExample` и дважды щелкните на правой колонке свойства `OnActivate` на вкладке `Events` этого окна.

Целые типы

Целые типы используются для хранения и преобразования целых чисел. В Delphi предусмотрено несколько целочисленных типов, отличающихся диапазоном возможных значений. В этой главе мы будем использовать тип `Integer`, занимающий в памяти 4 смежных байта и предназначенный для хранения целых чисел в диапазоне от $-2\,147\,483\,648$ до $+2\,147\,483\,647$. Более полные сведения о целых типах вы найдете в главе 6.

Над целыми числами определены следующие математические операции:

- `+` — сложение;
- `-` — вычитание;
- `*` — умножение;
- `div` — деление с отбрасыванием остатка;
- `mod` — получение остатка от деления.

Спецификой деления является то обстоятельство, что результат может иметь дробный вид: $\frac{1}{2}$, $2\frac{3}{5}$ и т. п. Для хранения дробных чисел в Delphi используются вещественные типы (см. главу 6), вот почему в языке поддерживаются целых две операции деления (`div` и `mod`):

```
var
  X, Y: Integer;
begin
  X := 5 div 2; // X содержит 2
  Y := 5 mod 2; // Y содержит 1
end;
```

Смысл остальных операций совпадает с общепринятым математическим, за тем исключением, что результат не должен выходить из диапазона возможных значений целого типа:

```
var
  X: Integer;
begin
  X := 2147483647;
  X := X + 1; // Ошибка! Результат +2 147 483 648 выходит
              // из допустимого диапазона
end;
```

Как и к строкам, к целым числам применимы операции сравнения.

Создадим программу, в которой пользователь вводит два целых числа, а программа вычисляет и показывает их произведение. В этой программе мы также познакомимся с преобразованием типов. На рис. 4.5 показано окно работающей программы.

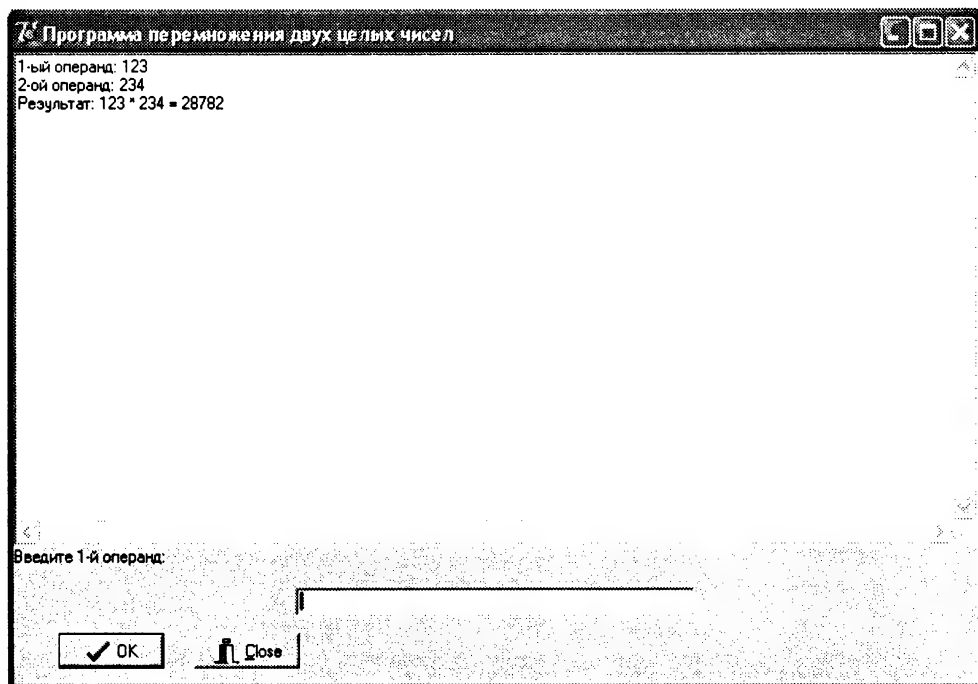


Рис. 4.5. Окно программы MultInt

Прежде всего обсудим две проблемы, связанные с реализацией программы:

- как известить программу о вводе каждого из операндов;
- что будет, если пользователь подготовит неверную запись целого числа (например, «12.3»).

Суть первой проблемы состоит в том, что подготовленная нами учебная форма `fmExample` имеет только одно поле ввода `edInput` и одну кнопку `bbRun` (кнопка `bbClose` используется для прекращения работы программы и эту ее функциональность изменять не следует). Мы могли бы написать программу, которая вводит строку вроде `2 * 3` и после анализа строки выводит результат. Однако логика анализа существенно перегрузит программу и потребует использования в ней средств, с которыми мы пока не знакомы. Значительно проще вставить в форму еще одну кнопку. Тогда щелчок на первой кнопке будет сигналом о том, что в строке ввода подготовлен первый операнд, а щелчок на второй — что готов второй операнд и можно вычислять результат. В этом варианте следует предусмотреть случай, когда щелчок на второй кнопке будет предшествовать щелчку на первой, и программа будет вычислять выражение с неопределенным первым операндом. Решение этой дополнительной проблемы состоит в том, чтобы сделать вторую кнопку недоступной или даже невидимой до тех пор, пока не будет введен первый операнд.

Подводя итог, я предлагаю поместить на форму `fmExample` новую кнопку `BitBtn` (вкладка `Additional` галереи компонентов) так, чтобы она полностью *закрыла собой* имеющуюся кнопку `bbRun`, назвать новую кнопку `bbResult` (свойство `Name`), уста-

новить для нее сорт (свойство `Kind`) `bkOK` и сделать ее временно невидимой (поместить в свойство `Visible` значение `False`). После такой подготовки в момент начала работы новая кнопка не будет видна и не помешает пользователю щелкнуть на кнопке `bbRun`. В обработчике события `bbRunClick` следует спрятать кнопку `bbRun` и показать кнопку `bbResult`, чтобы пользователь смог подтвердить ввод второго операнда. Сделайте необходимые изменения формы для нового проекта, а я пока перейду к обсуждению второй проблемы.

Суть второй проблемы заключается в том, что пользователь может случайно или преднамеренно ввести строку, которая не содержит правильного представления целого числа. Преобразование такой строки к целому числу может привести к аварийному завершению работы программы. Существует множество способов решения этой проблемы, я предлагаю наиболее простой: заменить обычное однострочное текстовое поле `edInput` специализированным, способным контролировать ввод пользователя. Удалите из формы поле `edInput` (щелкните на нем мышью и нажмите клавишу `Delete`) и поместите на его место компонент `MaskEdit` (вкладка `Additional`). Дайте ему прежнее имя `edInput` (свойство `Name`) и раскройте диалоговое окно свойства `EditMask` (рис. 4.6). Это свойство позволяет указать маску компонента, то есть определить, какие символы и в какой последовательности может набирать пользователь в поле.

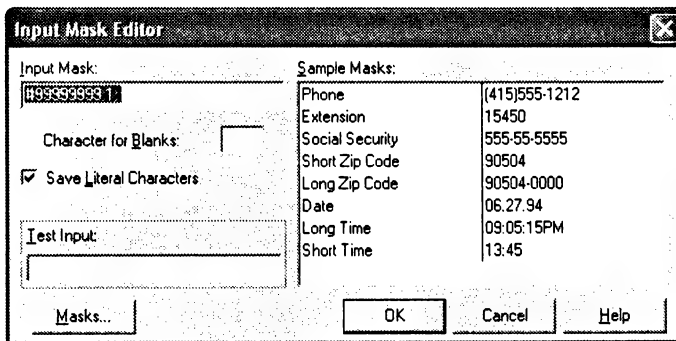


Рис. 4.6. Окно определения маски компонента `MaskEdit`

В поле `Input Mask` введите следующую последовательность символов:

```
#99999999;1;
```

В поле `Character for Blanks` вместо умалчиваемого символа-заполнителя, которым является символ подчеркивания (`_`), укажите пробел и закройте окно щелчком на кнопке `OK`. Сформированная нами маска заставит компонент автоматически контролировать ввод пользователя и отвергать любые символы, кроме знака `+` или `-` в первой позиции (элемент маски «`#`») и цифр в остальных (элементы «`9`»).

Теперь все готово к кодированию программы.

Дважды щелкните на кнопке `bbResult` и введите такой код для обработчика события `OnClick` этой кнопки:

```
procedure TfmExample.bbResultClick(Sender: TObject);
// Обработка ввода 2-го операнда
```

begin

```
// Преобразуем текст из edInput в целое число:
Y := StrToInt(Trim(edInput.Text));
// Сообщаем в Метод о вводе 2-го операнда:
mmOutput.Lines.Add('2-ой операнд: '+edInput.Text);
// Вычисляем и показываем результат:
mmOutput.Lines.Add('Результат: '+
  IntToStr(X)+' * '+IntToStr(Y)+' = '+IntToStr(X*Y));
edInput.Text := '';
edInput.SetFocus;
```

фокус

```
lbOutput.Caption := 'Введите 1-й операнд: ';
bbResult.Hide;
bbRun.Show;
```

end;

Для ввода обработчика события OnClick кнопки bbRun нужно сначала с помощью списка в верхней части окна инспектора объектов отыскать и выбрать компонент bbRun (напомню, он полностью закрыт кнопкой bbResult), затем дважды щелкнуть на правом столбце строки OnClick на вкладке Events и ввести такой код:

```
procedure TfmExample.bbRunClick(Sender: TObject);
// Обработка ввода 1-го операнда
begin
  // Преобразуем текст из edInput в целое число
  X := StrToInt(Trim(edInput.Text));
  // Сообщаем в Метод о вводе 1-го операнда
  mmOutput.Lines.Add('1-ый операнд: '+edInput.Text);
  edInput.Text := '';
  edInput.SetFocus;
  lbOutput.Caption := 'Введите 2-й операнд: ';
  bbResult.Show;
bbResult
  bbRun.Hide;
```

end;

Осталось передать фокус вводу компоненту edInput в момент старта программы и определить переменные X и Y для хранения операндов. Разыщите в самом начале окна кода программы описание класса TfmExample и отредактируйте его следующим образом:

```
private
  { Private declarations }
  X, Y: Integer;
```

Затем с помощью списка в окне инспектора объектов выберите форму fmExample и после двойного щелчка на строке OnActivate вкладки Events введите такой обработчик события:

```
procedure TfmExample.FormActivate(Sender: TObject);
// Подготовка к началу работы
```

```
begin
```

```
  edInput.SetFocus; // Передаем фокус ввода полю edInput
  lbOutput.Caption := 'Введите 1-й операнд:';
```

```
end;
```

С помощью следующих строк мы определили в классе `TfmExample` две целочисленные переменные `X` и `Y`:

```
private
```

```
  { Private declarations }
  X, Y: Integer;
```

После определения переменные становятся доступны любым методам этого класса, в том числе обработчикам событий `OnClick`. Такого рода переменные (их называют *полями* класса) облегчают взаимодействие методов друг с другом. В нашем случае необходимость в них возникла по той причине, что первый операнд (переменная `X`) вводится в одном методе (`bbRunClick`), а используется в другом (`bbResultClick`).

Оба обработчика событий `OnClick` содержат почти одинаковые операторы. В начале каждого обработчика стоит оператор присваивания, в правой части которого содержится выражение:

```
StrToInt (Trim (edInput.Text) )
```

Это выражение состоит из вызова стандартных для Delphi функций: сначала вызывается функция `Trim`, которая возвращает строку-аргумент (в нашем случае `edInput.Text`) без обрамляющих ее (то есть ведущих и ведомых) пробелов; затем с помощью вызова `StrToInt` строка символов преобразуется к целому числу.

Функция преобразования `StrToInt` очень чувствительна к возможным ошибкам символьного представления целого числа. Большая часть возможных ошибок в нашей программе блокируется фильтрующими возможностями поля `MaskEdit` и функцией `Trim`. Программа аварийно завершится, если при щелчке на кнопке `bbRun` или `bbResult` текстовое поле не содержит текста. Чтобы избежать этого, перед вызовом `StrToInt` вставьте следующий фрагмент:

```
if edInput.Text = '' then Exit;
```

Этот оператор прекращает дальнейшую работу обработчика в случае, если в поле ввода `edInput` нет текста.

Методы `Show` и `Hide` компонентов-кнопок соответственно показывают или прячут компонент. Спрятанный компонент никак не участвует во взаимодействии с пользователем.

Операторы языка

С одним из наиболее часто используемых операторов языка Delphi — оператором присваивания — мы уже познакомились. Ниже рассматриваются остальные операторы.

Составной оператор и пустой оператор

Составной оператор — это последовательность произвольных операторов программы, заключенная в операторные скобки — зарезервированные слова **begin** и **end**.

Составные операторы — важный инструмент Delphi, дающий возможность писать программы по современной технологии структурного программирования (без операторов перехода **goto**).

Delphi не накладывает никаких ограничений на характер операторов, входящих в составной оператор. Среди них могут быть и другие составные операторы — язык Delphi допускает произвольную глубину их вложенности:

```
begin
  ...
  begin
    ...
    begin
      ...
      ...
    end;
  ...
end;
...
end;
```

Фактически, весь раздел операторов, обрамленный словами **begin** и **end**, представляет собой один составной оператор. Поскольку зарезервированное слово **end** является закрывающей операторной скобкой, оно одновременно указывает и на конец предыдущего оператора, поэтому ставить перед ним символ точки с запятой (;) не обязательно. Наличие точки с запятой перед **end** в предыдущих примерах означало, что между последним оператором и операторной скобкой **end** располагается *пустой оператор*. Пустой оператор не выполняет никаких действий, просто в программу добавляется лишняя точка с запятой. В основном пустой оператор используется для передачи управления в конец составного оператора: как и любой другой, пустой оператор может быть помечен, и ему можно передать управление.

Условный оператор

Условный оператор позволяет проверить некоторое условие и в зависимости от результатов проверки выполнить то или иное действие. Таким образом, условный оператор — это средство ветвления вычислительного процесса.

Структура условного оператора имеет следующий вид:

```
if <условие> then <оператор1> else <оператор2>;
```

Здесь **if**, **then**, **else** — зарезервированные слова (*если, то, иначе*); <условие> — произвольное выражение логического типа; <оператор1>, <оператор2> — любые операторы языка Delphi.

ПРИМЕЧАНИЕ

После оператора <оператор1> ставить признак конца оператора (символ точки с запятой) нельзя, если следом за ним идет зарезервированное слово **else**.

Условный оператор работает по следующему алгоритму. Вначале вычисляется условное выражение <условие>. Если результат есть True (истина), то выпол-

няется <оператор1>, а <оператор2> пропускается; если результат есть False (ложь), наоборот, <оператор1> пропускается, а выполняется <оператор2>. Например:

```
var
  X, Y, Max: Integer;
begin
  ...
  if X > Max then
    Y := Max
  else
    Y := X;
  ...
end;
```

При выполнении этого фрагмента переменная Y получит значение переменной X, если только это значение не превышает Max, в противном случае Y станет равно Max.

Условными называются выражения, имеющие одно из двух возможных значений: истина или ложь. Такие выражения чаще всего получаются при сравнении переменных в операциях отношения =, <>, >, >=, <, <=. Сложные логические выражения составляются с использованием логических операций **and** (логическое И), **or** (логическое ИЛИ) и **not** (логическое НЕ). Например:

```
if (a > b) and (b <> 0) then ...
```

ВНИМАНИЕ В отличие от большинства других языков программирования, в Delphi приоритет операций отношения меньше, чем у логических операций, поэтому отдельные составные части сложного логического выражения заключаются в скобки. Например, такая запись предыдущего оператора будет неверной:

```
if a > b and b <> 0 then ... // Ошибка!
```

Так как фактически (с учетом приоритета операций) компилятор будет транслировать такую строку:

```
if a > (b and b) <> 0 then ...
```

Часть **else** <оператор2> условного оператора может быть опущена. Тогда при значении True условного выражения выполняется <оператор1>, в противном случае этот оператор пропускается:

```
var
  X, Y, Max: Integer;
begin
  ...
  if X > Max then
    Max := X;
  Y := X;
  ...
end;
```


В этом примере переменная *Y* всегда будет иметь значение переменной *X*, а в *Max* запоминается максимальное значение *X*.

Поскольку любой из операторов *<оператор1>* и *<оператор2>* может быть любым, в том числе и условным, а в то же время не каждый из «вложенных» условных операторов может иметь часть **else** *<оператор2>*, то возникает неоднозначность трактовки условий. Эта неоднозначность в Delphi решается следующим образом: любая встретившаяся часть **else** соответствует ближайшей к ней сверху по тексту программы части **then** условного оператора. Например:

```
var
  a,b,c,d : Integer;
begin
  a := 1;
  b := 2;
  c := 3;
  d := 4;
  if a < b then                                // Да
    if c < d then                              // Да
      if c < 0 then                          // Нет
        c := 0                               // Этот оператор не
        ВЫПОЛНЯЕТСЯ
      else
        a := b;                             // a равно 2
    if a < b then                              // Да
      if c < d then                          // Да
        if c < 0 then                      // Нет
          c := 0                            // c равно 0
        else                                // if c < 0
          else                              // if c < d
            else                            // if a < b
              a := b;                       // Этот оператор не
              ВЫПОЛНЯЕТСЯ
end;
```

Создадим игру «Угадай число»: программа случайным образом выбирает целое число в диапазоне 0...1000 и запоминает его; пользователь пытается угадать его, вводя свое число. Сравнив ввод с запомненным числом, программа сообщает больше, меньше или равно введенное число запомненному. Ввод продолжается до угадывания, после чего программа предлагает сыграть еще раз.

Поскольку пользователь вводит только числа, изменим форму *fmExample*: как и в предыдущей программе *MULTINT* вместо компонента *edInput* типа *TEdit* поместим на форму одноименный компонент типа *TMaskEdit*. Выберите команду **File** ► **New** ► **Application**, щелкните на компоненте *edInput* и удалите его с помощью клавиши **Delete**, затем поместите на его место компонент *MaskEdit* (вкладка **Additional**), назовите его *edInput* (свойство *Name*) и в его свойство *EditMask* поместите строку:

```
0999;1;
```

В секцию **private** описания класса `fmExample` поместите объявление целочисленного поля `X`:

```
private
    { Private declarations }
    X: Integer;
```

Для обработчика события `OnActivate` формы `fmExample` напишите такой код:

```
procedure TfmExample.FormActivate(Sender: TObject);
begin
    X := Random(1001);    // Загадываем случайное число
    edInput.SetFocus;     // Передаем полю edInput фокус ввода
    Caption := 'Угадай целое число в диапазоне 0...1000';
    lbOutput.Caption := 'Введите число: ';
end;
```

Поместите в обработчик события `OnClick` кнопки `bbRun` такой код:

```
procedure TfmExample.bbRunClick(Sender: TObject);
var
    Y: Integer;
begin
    if edInput.Text = '' then
        Exit;    // Если нет входного текста, прекращаем работу
    // Преобразуем ввод в число:
    Y := StrToInt(Trim(edInput.Text));
    edInput.Text := '';    // Очищаем поле ввода
    edInput.SetFocus;     // Устанавливаем фокус ввода
    { Параметр Tag = 0 означает угадывание числа, иначе -
      ответ на вопрос, будет ли пользователь играть после
      угадывания }
    if Tag = 0 then
        if X < Y then    // Угадывание числа
            mmOutput.Lines.Add('X < '+IntToStr(Y))
        else if X > Y then
            mmOutput.Lines.Add('X > '+IntToStr(Y))
        else    // X = Y
            begin
                mmOutput.Lines.Add('X = '+IntToStr(Y));
                MessageBeep(MB_OK); // Звук для привлечения внимания
                lbOutput.Caption :=
                    'Вы угадали! Введите 1, если хотите повторить: ';
                Tag := 1;    // Следующий ввод - ответ на вопрос
            end
        else    // Анализ ответа
            if Y = 1 then
                begin
                    X := Random(1001); // Новое число
```

```

lbOutput.Caption := 'Введите число: ';
Tag := 0;           // Следующий ввод - угадывание
edInput.SetFocus
end else
  Close;
end;

```

При активизации окна программы обработчик `FormActivate` с помощью следующего оператора помещает в поле `X` случайное целое число в диапазоне от 0 до $1001 - 1 = 1000$ (параметр обращения к генератору псевдослучайных чисел `Random` на единицу больше верхней границы диапазона):

```
X := Random(1001);
```

Каждый компонент Delphi имеет целочисленное свойство `Tag`, которым программист может распоряжаться по своему усмотрению. Программа использует это свойство формы `fmExample` для разделения алгоритма обработки события `OnClick` кнопки `bbRun` на два варианта: пользователь угадывает число (`Tag = 0`) или отвечает на предложение сыграть еще раз (`Tag = 1`). В момент создания экземпляра класса `TfmExmple` — объекта `fmExample` — все его поля обнуляются, поэтому первоначально `Tag = 0`, и программа анализирует угадывание числа. При угадывании срабатывает составной оператор:

```

begin
  mmOutput.Lines.Add('X = '+IntToStr(Y));
  MessageBeep(MB_OK); // Звук для привлечения внимания
  lbOutput.Caption :=
    'Вы угадали! Введите 1, если хотите повторить: ';
  Tag := 1;           // Следующий ввод - ответ на вопрос
end

```

После этого `Tag` получает значение 1 (стандартная процедура `MessageBeep` заставляет звуковую систему ПК издать короткий звуковой сигнал). Теперь очередной щелчок на кнопке `bbRun` будет означать, что пользователь подготовил ответ на вопрос, и программа анализирует его. Если пользователь ввел число, отличное от 1, произойдет обращение к методу `Close` формы `fmExample` и программа прекратит работу.

Операторы повторений

В языке Delphi имеются три различных оператора, с помощью которых можно запрограммировать повторяющиеся фрагменты программ.

Счетный оператор цикла `for` имеет такую структуру:

```

for <параметр_цикла> := <нач_знач> to <кон_знач>
do <оператор>;

```

Здесь **for**, **to**, **do** — зарезервированные слова (*для, до, выполнить*); `<параметр_цикла>` — переменная типа `Integer` (точнее, любого порядкового типа, см. главу 6); `<нач_знач>` — начальное значение (выражение того же типа); `<кон_знач>` — конечное значение (выражение того же типа); `<оператор>` — произвольный оператор Delphi.

При выполнении оператора **for** вначале вычисляется выражение <нач_знач> и осуществляется присваивание <параметр_цикла> := <нач_знач>. После этого циклически повторяется следующая последовательность действий.

1. Проверка условия <параметр_цикла> <= <кон_знач>; если условие не выполнено, оператор **for** завершает свою работу.
2. Выполнение оператора <оператор>.
3. Нарастивание переменной <параметр_цикла> на единицу.

В качестве иллюстрации применения оператора **for** рассмотрим программу IntSumm, осуществляющую ввод произвольного целого числа N и вычисление суммы всех целых чисел от 1 до N.

Для нового приложения (команда File ► New ► Application) создайте такой обработчик bbRunClick:

```
procedure TfmExample.bbRunClick(Sender: TObject);
var
    i, N, Sum : Integer;
begin
    try          // Преобразуем ввод с контролем правильности:
        N := StrToInt(edInput.Text);
    except      // Следующие операторы выполняются, если есть ошибка
        ShowMessage('Ошибка ввода целого числа');
        edInput.SelectAll;  // Выделяем неверный ввод
        Exit                // Завершаем работу обработчика
    end;
    edInput.Text := '';
    edInput.SetFocus;
    Sum := 0;                // Начальное значение Sum
    for i := 1 to N do      // Цикл формирования суммы
        Sum := Sum+i;
    mmOutput.Lines.Add('Сумма всех целых чисел ' +
        'в диапазоне 1...' + IntToStr(N) + ' равна '
        + IntToStr(Sum));
end;
```

Прежде всего, обратите внимание на операторы:

```
try          // Преобразуем ввод с контролем правильности:
    N := StrToInt(edInput.Text);
except      // Следующие операторы выполняются,
            // если есть ошибка
    ShowMessage('Ошибка ввода целого числа');
    edInput.SelectAll;  // Выделяем неверный текст
    Exit                // Завершаем работу обработчика
end;
```

С помощью зарезервированных слов **try** (попробовать), **except** (исключение) и **end** реализуется так называемый *защищенный блок*. Такими блоками программист может защитить программу от краха при выполнении потенциально опасно-

го участка (подробнее см. главу 13). В отличие от предыдущих программ, мы не изменили компонент `edInput`, поэтому пользователь может ввести в нем произвольный текст. Если этот текст не содержит правильное представление целого числа, попытка выполнить следующий оператор в обычной программе привела бы к аварийному завершению программы:

```
N := StrToInt(edInput.Text);
```

Чтобы этого не произошло, мы защитили этот оператор, расположив его за словом **try** и перед словом **except**. Если ошибки нет, все операторы, стоящие за **except** и перед **end**, пропускаются, и обработчик нормально срабатывает. Если обнаружена ошибка, возникает так называемая *исключительная ситуация* (исключение), и управление автоматически передается оператору, стоящему первым за словом **except**, — начинается обработка исключения. Вначале с помощью стандартной процедуры `ShowMessage` мы сообщаем пользователю об ошибке¹, затем с помощью оператора `edInput.SelectAll` выделяем ошибочный текст в компоненте `edInput` и, наконец, путем вызова стандартной процедуры `Exit` аварийно завершаем работу обработчика (но не программы!).

Отметим также два обстоятельства. Во-первых, условие, управляющее работой оператора **for**, проверяется *перед* выполнением оператора <оператор>: если условие не выполняется в самом начале работы оператора **for**, исполняемый оператор не будет выполнен ни разу. Другое обстоятельство — шаг наращивания параметра цикла строго постоянен и равен +1. Существует другая форма оператора:

```
for <пар_цикл> := <нач_знач> downto <кон_знач> do
  <оператор>;
```

Замена зарезервированного слова **to** на **downto** означает, что шаг наращивания параметра цикла равен -1, а управляющее условие приобретает вид <параметр_цикла> >= <кон_знач>.

Предыдущий пример можно модифицировать так, чтобы сделать его пригодным для подсчета любых сумм — положительных и отрицательных:

```
...
Sum := 0;
if N >= 0 then
  for i := 1 to N do
    Sum := Sum + i
else
  for i := -1 downto N do
    Sum := Sum + i;
...

```

Два других оператора повторений лишь проверяют условие выполнения или повторения цикла, но не связаны с изменением счетчика цикла.

Оператор цикла while с предварительной проверкой условия:

¹ Если вы запустите программу из среды Delphi, исключение будет сначала перехвачено средой, и на экране появится сообщение на английском языке о характере и месте возникновения ошибки. В этом случае закройте окно с сообщением и нажмите клавишу F9 — программа продолжит свою работу, и вы увидите окно процедуры `ShowMessage`.

while <условие> **do** <оператор>;

Здесь **while**, **do** — зарезервированные слова (пока [выполняется условие], делать); <условие> — выражение логического типа; <оператор> — произвольный оператор Delphi.

Если выражение <условие> имеет значение True, то выполняется <оператор>, после чего вычисление выражения <условие> и его проверка повторяются. Если <условие> имеет значение False, оператор **while** прекращает свою работу.

Разработаем учебную программу, отыскивающую значение так называемого «машинного эпсилон» — минимального не равного нулю вещественного числа, которое после прибавления его к 1,0 еще дает результат, отличный от 1,0. Замечу, что для хранения и преобразования дробных чисел в Delphi предназначены так называемые вещественные типы (см. главу 6). В учебной программе используется один из этих типов — Real, занимающий 8 смежных байтов и представляющий дробные (вещественные) числа в диапазоне от 10^{-324} до 10^{+308} с точностью 14...16 значащих цифр.

У читателя, привыкшего к непрерывной вещественной арифметике, может вызвать недоумение утверждение о том, что в дискретной машинной арифметике всегда существуют такие числа $0 < X < \text{eps}$, что $1,0 + X = 1,0$. Дело в том, что внутреннее представление типа Real может дать «лишь» приблизительно 10^{19} возможных комбинаций значащих разрядов в отведенных для него 8 байтах. Конечно же, это очень большое число, но оно несопоставимо с бесконечным множеством вещественных чисел. Аппроксимация бесконечного непрерывного множества вещественных чисел конечным (пусть даже и очень большим) множеством их внутреннего машинного представления и приводит к появлению «машинного эпсилон».

Для нового приложения (команда File ► New ► Application) создайте такой обработчик bbRunClick:

```
procedure TfmExample.bbRunClick(Sender: TObject);
var
    Epsilon: Real;
begin
    Epsilon := 1;
    while 1+Epsilon/2>1 do
        Epsilon := Epsilon/2;
    lbOutput.Caption := 'Машинное эпсилон = '
        + FloatToStr(Epsilon)
end;
```

Для вещественных чисел можно использовать операцию деления без отбрасывания дробной части (знак /). После применения этой операции результат всегда имеет вещественный тип, поэтому такой фрагмент программы ошибочен:

```
var
    X: Integer;
begin
    X := 4/2; // Ошибка! Вещественный результат нельзя
              // присвоить целой переменной
end;
```

Стандартная функция `FloatToStr` преобразует вещественное число в строку символов.

*Оператор цикла **repeat ... until** с последующей проверкой условия:*

```
repeat <тело_цикла> until <условие>;
```

Здесь **repeat** и **until** — зарезервированные слова (повторять [до тех пор], пока [не будет выполнено условие]); <тело_цикла> — произвольная последовательность операторов Delphi; <условие> — выражение логического типа.

Операторы <тело_цикла> выполняются хотя бы один раз, после чего вычисляется выражение <условие>: если его значение есть `False`, операторы <тело_цикла> повторяются, в противном случае оператор **repeat ... until** завершает свою работу.

Обратите внимание: пара слов **repeat** и **until** подобна операторным скобкам **begin** и **end**, поэтому перед **until** ставить точку с запятой не обязательно.

Замечу, что для правильного выхода из цикла условие выхода должно меняться внутри операторов, составляющих тело цикла **while** или **repeat ... until**. Следующие циклы никогда не завершатся «естественным» способом:

```
while True do
begin
    ...
end;
repeat
    ...
until False;
```

Для гибкого управления циклическими операторами **for**, **while** и **repeat** в состав Delphi включены две процедуры без параметров:

- **BREAK** — реализует немедленный выход из цикла; действие процедуры заключается в передаче управления оператору, стоящему сразу за концом циклического оператора;
- **CONTINUE** — обеспечивает досрочное завершение очередного прохода цикла; эквивалент передачи управления в самый конец циклического оператора.

Введение в язык этих процедур практически исключает необходимость использования операторов безусловного перехода **goto** (см. ниже подраздел «Метки и операторы перехода»).

Оператор выбора

Оператор выбора позволяет выбрать одно из нескольких возможных продолжений программы. Параметром, по которому осуществляется выбор, служит *ключ выбора* — выражение любого порядкового типа (из уже рассмотренных к порядковым относятся `Integer`, `Char` и логический тип; описание остальных порядковых типов см. в главе 6).

Структура оператора выбора такова:

```
case <ключ_выбора> of <список_выбора> [else <операторы>]
end;
```

Здесь **case, of, else, end** — зарезервированные слова (*случай, из, иначе, конец*); <ключ_выбора> — ключ выбора (выражение порядкового типа); <список_выбора> — одна или более конструкций вида:

<константа_выбора>: <оператор>;

Здесь <константа_выбора> — константа того же типа, что и выражение <ключ_выбора>; <оператор> — произвольный оператор Delphi.

Оператор выбора работает следующим образом. Вначале вычисляется значение выражения <ключ_выбора>, а затем в последовательности операторов <список_выбора> отыскивается такой, которому предшествует константа, равная вычисленному значению. Найденный оператор выполняется, после чего оператор выбора завершает свою работу. Если в списке выбора не будет найдена константа, соответствующая вычисленному значению ключа выбора, управление передается операторам, стоящим за словом **else**. Часть **else** <операторы> можно опускать. Тогда при отсутствии в списке выбора нужной константы ничего не произойдет, и оператор выбора просто завершит свою работу.

Любому из операторов списка выбора может предшествовать не одна, а несколько констант выбора, разделенных запятыми. Например:

```
var
  ch : Char;
begin
  .....
  case ch of
    'n', 'N', 'н', 'Н': lbOutput.Caption := 'Нет';
    'y', 'Y', 'д', 'Д': lbOutput.Caption := 'Да';
  end
end;
```

Составим программу, имитирующую работу микрокалькулятора. Программа поддерживает ввод двух операндов и знака математического действия, после чего показывает результат (рис. 4.7).

Для упрощения ввода операндов и знака операции поместим на нашу учебную форму еще два компонента. Выберите команду File ► New ► Application, переименуйте компонент lbInput в lbInput1 и сделайте его свойство Width (длина) равным 121 (стандартная длина компонентов Edit), а свойство Left (слева) — 160. Справа от компонента lbInput1 поместите компонент ComboBox (вкладка Standard), назовите его cbSign (свойство Name) и установите для него значения свойств Width — 41 и Left — 328. Раскройте диалоговое окно свойства Items этого компонента и введите четыре строки со знаками математических действий в такой последовательности:

```
+
-
*
/
```

И, наконец, справа от компонента cbSign разместите еще один компонент Edit (вкладка Standard) и назовите его edInput2 (Left — 416). Не забудьте очистить его свойство Text.

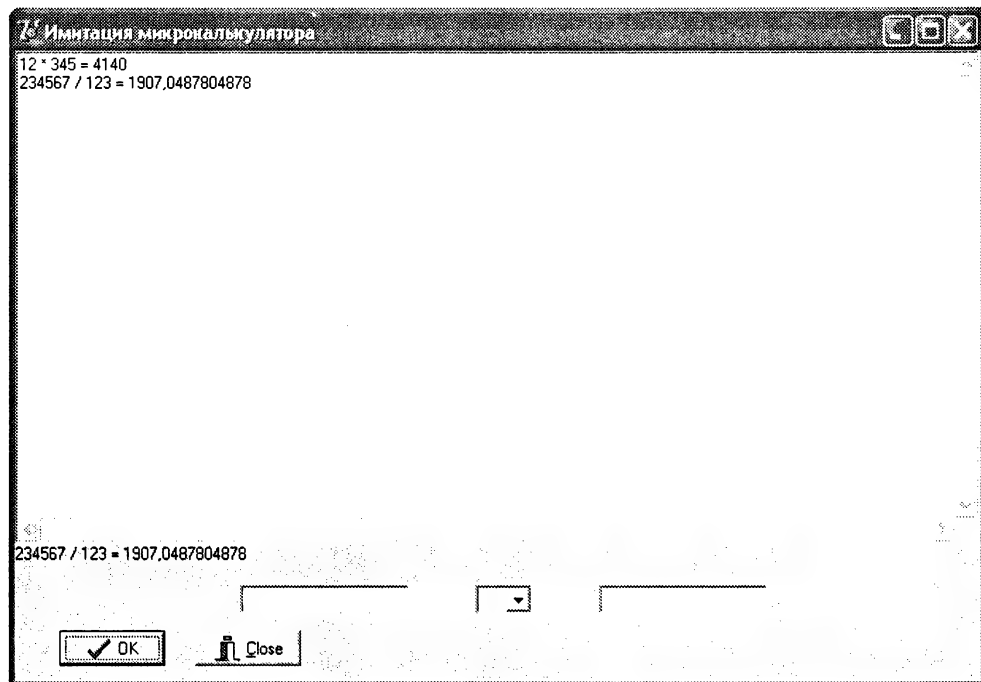


Рис. 4.7. Окно программы CALC

Для обработчика события `bbRunClick` введите такой текст:

```
procedure TfmExample.bbRunClick(Sender: TObject);
var
    X, Y, Z: Real;
begin
    { Блокируем работу, если не введены операнды и не выбрано
    действие }
    if (edInput1.Text='') or (edInput2.Text='')
        or (cbSign.ItemIndex<0) then Exit;
    // Проверяем правильность ввода 1-го операнда:
    try                // Начало защищенного блока
        X := StrToFloat(Trim(edInput1.Text));
    except             // Обработка ошибки
        ShowMessage('Ошибочная запись числа: '+edInput1.Text);
        edInput1.SelectAll;
        Exit;
    end;               // Конец защищенного блока
    // Проверяем правильность ввода 2-го операнда:
    try                // Начало защищенного блока
        Y := StrToFloat(Trim(edInput2.Text));
    except             // Обработка ошибки
        ShowMessage('Ошибочная запись числа: '+edInput2.Text);
```

```

    edInput2.SelectAll;
    Exit;
end;           // Конец защищенного блока
// Все правильно: вычисляем результат
case cbSign.ItemIndex of
0: Z := X + Y;
1: Z := X - Y;
2: Z := X * Y;
3: try
    Z := X / Y;
except
    Z := 1.1e+38;    // Бесконечность при делении на 0
end;
end;
// Показываем результат
lbOutput.Caption := Trim(edInput1.Text)+' '+
cbSign.Items[cbSign.ItemIndex]+' '+
Trim(edInput2.Text)+' = ';
if Z >= 1.1e+38 then
    lbOutput.Caption := lbOutput.Caption+'бесконечность'
else
    lbOutput.Caption := lbOutput.Caption+FloatToStr(Z);
mmOutput.Lines.Add(lbOutput.Caption);
// Очищаем ввод
edInput1.Text := '';
edInput2.Text := '';
cbSign.ItemIndex := -1;
end;

```

В качестве операндов в программе выбраны переменные вещественного типа Real. Символьное представление значений Real в общем случае имеет вид:

$\pm\text{ЦЦЦЦЦ}.\text{ДДДДД}e\pm\text{ППП}$

Здесь $\pm$ — знак + или –; ЦЦЦЦЦ — цифры целой части числа; ДДДДД — цифры дробной части; ППП — цифры десятичного порядка. Символ e (Е) в такой записи читается как «умножить на 10 в степени». Например, $1.1e+38$ читается: «один и одна десятая умножить на 10 в степени плюс тридцать восемь». Следует учесть, что компилятор Delphi в качестве разделителя целой и дробной частей всегда использует символ точки (.), в то время как аналогичный разделитель в Windows зависит от выбранной страны и для России им обычно является запятая (,). Это означает, что вещественные константы в *тексте программы* вы всегда должны записывать с точкой в качестве разделителя, а *при вводе* вещественных чисел пользователь программы должен указывать разделитель Windows (этот разделитель можно выбрать в 32-разрядных версиях Windows с помощью команды Пуск ► Настройка ► Панель управления и последующим заданием требуемых параметров в диалоговом окне Язык и стандарты).

Поскольку правильная форма записи вещественного числа гораздо богаче формы записи целого, использовать однострочное поле с маской для фильтрации вещественных чисел невозможно, поэтому в программе используются обычные однострочные поля `edInput1` и `edInput2`. В этой ситуации контроль за правильностью записи приходится возлагать на функцию преобразования `StrToFloat` (эта функция преобразует строку в вещественное число). Если функция обнаружит ошибку в записи вещественного числа, она инициирует исключительную ситуацию, которая обычно приводит к аварийному завершению работы программы. Чтобы этого не произошло, используются защищенные блоки (см. комментарий к программе `INTSUMM`). Таким же блоком защищается операция деления: если число делится на ноль или на очень малую величину, результату присваивается большая положительная величина 1.1×10^{38} . При выводе сообщения эта величина заменяется словом «бесконечность».

Для выбора операции в программе используется компонент `ComboBox` — раскрывающийся список. Свойство `Items` этого компонента хранит набор строк, которые определяют список. Эти строки пронумерованы, начиная от нуля (в нашем случае `cbSign.Items[0] = '+'`, а `cbSign.Items[3] = '/'`). Номер выбранной строки содержит свойство `cbSign.ItemIndex`, а если не выбрана ни одна строка, это свойство содержит `-1`. При проверке операндов программа контролирует выбор за счет условия:

```
cbSign.ItemIndex < 0
```

Кроме того, выбор контролируется в операторе **case**:

```
case cbSign.ItemIndex of
```

Если вы ввели строки `cbSign.Items` в нужной последовательности, индекс выбранной строки будет соответствовать номеру операции (`0 — +`, `1 — -`, `2 — *` и `3 — /`) и над операндами будет выполнено требуемое действие.

Метки и операторы перехода

Можно теоретически показать, что рассмотренных операторов вполне достаточно для написания программ любой сложности. В этом отношении наличие в языке операторов перехода кажется излишним. Более того, современная технология структурного программирования основана на принципе «программировать без **goto**»: считается, что злоупотребление операторами перехода затрудняет понимание программы, делает ее запутанной и сложной в отладке.

Тем не менее, в некоторых случаях использование операторов перехода может упростить программу.

Оператор перехода имеет вид:

```
goto <метка>;
```

Здесь **goto** — зарезервированное слово (перейти [на метку]); `<метка>` — метка.

Метка в Delphi — это произвольный идентификатор, позволяющий именовать некоторый оператор программы и таким образом ссылаться на него. В целях совместимости со стандартным языком Паскаль в Delphi допускается в качестве меток использование также целых чисел без знака.

Метка располагается непосредственно перед помечаемым оператором и отделяется от него двоеточием. Оператор можно помечать несколькими метками, которые в этом случае отделяются друг от друга двоеточием. Перед тем как появиться в программе, метка должна быть описана. Описание меток состоит из зарезервированного слова **label** (метка), за которым следует список меток:

```
label
  loop, lb1, lb2;
begin
  ...
  goto lb1;
  ...
  loop: ...
  ...
  lb1:lb2: ...
  ...
  goto lb2;
  ...
end;
```

Действие оператора **goto** состоит в передаче управления соответствующему «меченному» оператору.

При использовании меток необходимо руководствоваться следующими правилами:

- метка, на которую ссылается оператор **goto**, должна быть описана в разделе описаний и встречаться где-нибудь еще в теле программы;
- метки, описанные в подпрограмме, локализируются в ней, поэтому передача управления извне подпрограммы на метку внутри нее невозможна.

Массивы

Рассмотренные выше простые типы данных позволяют использовать в программе одиночные объекты — числа, символы, строки и т. п. В Delphi могут также применяться объекты, содержащие множество однотипных элементов. Это *массивы* — формальное объединение нескольких однотипных объектов (чисел, символов, строк и т. п.), рассматриваемое как единое целое. К необходимости применения массивов мы приходим всякий раз, когда требуется связать и использовать целый ряд родственных величин. Например, результаты многократных замеров температуры воздуха в течение года удобно рассматривать как совокупность вещественных чисел, объединенных в один сложный объект — массив измерений.

При описании массива необходимо указать общее количество входящих в массив элементов и тип этих элементов. Например:

```
var
  a: array [1..10] of Real;
  b: array [0..50] of Char;
  c: array [-3..4] of String;
```

Как видим, при описании массива используются зарезервированные слова **array** и **of** (массив¹, из). За словом **array** в квадратных скобках указывается тип-диапазон, с помощью которого компилятор определяет общее количество элементов массива. Тип-диапазон (подробнее см. в главе 6) задается левой и правой границами изменения индекса массива, так что массив A состоит из 10 элементов, массив B — из 51, а массив C — из 8 элементов. За словом **of** указывается тип элементов, образующих массив.

Доступ к каждому элементу массива в программе осуществляется по *индексу* — целому числу (точнее, выражению порядкового типа, см. главу 6), служащему своеобразным именем элемента в массиве (если левая граница типа-диапазона равна 1, индекс элемента совпадает с его порядковым номером). При упоминании в программе любого элемента массива сразу за именем массива должен следовать индекс элемента в квадратных скобках, например:

```
var
  a: array [1..10] of Integer;
  b: array [0..40] of Char;
  c: array [-2..2] of Boolean;
  k: Integer;
begin
  ...
  b[17] := 'F';
  c[-2] := a[1] > 2;
  for k := 1 to 10 do
    a[k] := 0;
  ...
end;
```

В правильно составленной программе индекс не должен выходить за пределы, определенные типом-диапазоном. Например, можно использовать элементы A[1], B[38], C[0], но нельзя A[0] или C[38] (определение массивов см. выше). Компилятор Delphi может контролировать использование индексов в программе как на этапе ее компиляции, так и на этапе прогона.

Для иллюстрации приемов работы с массивами составим программу, которая создает массив случайных целых чисел, подсчитывает их среднее арифметическое, а также определяет и выводит на экран минимальное и максимальное из этих чисел.

```
procedure TfmExample.bbRunClick(Sender: TObject);
{Программа создает массив из N случайных целых чисел,
 равномерно распределенных в диапазоне от 0 до MAX_VALUE-1,
 подсчитывает среднее арифметическое этих чисел, а также
 минимальное и максимальное из них.}
const
  N = 1000;           // Количество элементов массива
  MAX_VALUE = 100+1; // Диапазон значений случайных чисел
```

¹ Буквальный перевод *атау* — боевой порядок, упорядоченная масса (войск). В компьютерной терминологии *атау* переводится словом *массив*.

```

var
  m: array [1..N] of Integer; // Массив чисел
  i: Integer;                 // Индекс массива
  max, min: Integer; // Максимальное и минимальное число
  sum: Real;                // Сумма чисел
  S: String;                // Буферная строка
begin
  S := '';
  // Наполняем массив случайными числами:
  for i := 1 to N do
    begin
      m[i] := Random(MAX_VALUE);
      S := S+#9+IntToStr(m[i]); // Добавляем число к буферу
      if i mod 10=0 then        // - В буфере 10 чисел?
        begin                  // - Да. Выводим буфер
          mmOutput.Lines.Add(S);
          S := ''
        end
      end;
    // Задаем начальные значения переменных:
    sum := m[1];
    max := m[1];
    min := m[1];
  // Цикл вычисления суммы всех случайных чисел и поиска
  // минимального и максимального:
  for i := 2 to N do
    begin
      sum := sum + m[i];
      if m[i] < min then
        min := m[i]
      else if m[i] > max then
        max := m[i]
      end;
    // Вычисляем среднее значение и выводим результат:
    lbOutput.Caption := 'Мин = '+IntToStr(min)+' Макс = '+
      IntToStr(max)+' Среднее = '+FloatToStr(sum/N);
  end;
end;

```

Для создания массива используется встроенная функция `Random (Max)`, которая возвращает случайное целое число, равномерно распределенное в диапазоне от 0 до `Max - 1` (`Max` — параметр обращения). Каждое случайное число переводится в строковый формат и добавляется к буферной строке (символы #9 определяют вставку табулостопа, разделяющего соседние числа в строке) следующим оператором:

```

S := S+#9+IntToStr(m[i]); // Добавляем число к буферу

```

После накопления в буферной строке 10 чисел строка помещается в многострочное поле `tmOutput` и очищается. Таким способом экономится время вывода и упорядочивается форма представления чисел на экране.

Процедуры и функции

Процедуры и функции (я часто буду использовать их общее название — *подпрограммы*) представляют собой важный инструмент Delphi, позволяющий писать хорошо структурированные программы. В структурированных программах обычно легко прослеживается основной алгоритм, их проще понять любому читателю, они удобнее в отладке и менее чувствительны к ошибкам программирования. Все эти свойства являются следствием важной особенности подпрограмм, каждая из которых представляет собой во многом самостоятельный фрагмент программы, связанный с основной программой лишь с помощью нескольких параметров. Самостоятельность подпрограмм позволяет локализовать в них все детали программной реализации того или иного алгоритмического действия и поэтому изменение этих деталей, например в процессе отладки, обычно не приводит к изменениям основной программы.

Многие примеры программ в этой книге невелики по размерам (не более 30–40 строк), поэтому в таких программах можно обойтись и без подпрограмм. Иное дело — создание крупных программ в сотни, тысячи и десятки тысяч строк. Писать такие программы как нечто единое целое без расчленения на относительно самостоятельные фрагменты, то есть без *структурирования*, просто невозможно. Практически во всех языках программирования имеются средства структурирования. Языки, в которых предусмотрены такие механизмы, называются *процедурно-ориентированными*. К их числу принадлежит и Delphi.

Процедурой в Delphi называется особым образом оформленный фрагмент программы, имеющий собственное имя. Упоминание этого имени в тексте программы приводит к активизации процедуры и называется ее *вызовом*. Сразу после активизации процедуры начинают выполняться входящие в нее операторы, после выполнения последнего из них управление возвращается обратно в основную программу и выполняются операторы, стоящие непосредственно за оператором вызова процедуры (рис. 4.8).

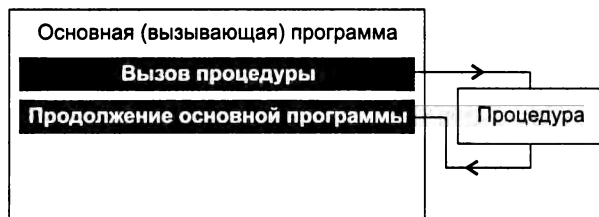


Рис. 4.8. Взаимодействие вызывающей программы и процедуры

Для обмена информацией между основной программой и процедурой используется один или несколько параметров вызова. Как мы увидим дальше (см. главу 10), процедуры могут иметь и другой механизм обмена данными с вызывающей

программой, так что параметры вызова могут и не использоваться. Если они есть, то перечисляются в круглых скобках за именем процедуры и вместе с ним образуют оператор вызова процедуры.

Функция отличается от процедуры тем, что результат ее работы возвращается в виде значения этой функции, и, следовательно, вызов функции может использоваться наряду с другими операндами в выражениях.

С примерами процедур и функций мы уже сталкивались — это стандартные процедуры `Exit`, `ShowMessage`, функции `StrToInt`, `FloatToStr`, `Random`, математические функции и др. Стандартными они называются потому, что созданы вместе с системой Delphi и являются ее неотъемлемой частью. В Delphi имеется много стандартных процедур и функций (см. описание некоторых из них в приложении Б). Наличие богатой библиотеки таких программных заготовок существенно облегчает разработку прикладных программ. Однако в большинстве случаев некоторые специфичные для данной прикладной программы действия не находят прямых аналогов в библиотеках Delphi, и тогда программисту приходится разрабатывать свои нестандартные процедуры и функции.

Нестандартную подпрограмму необходимо описать, чтобы компилятор смог установить связь между оператором вызова и теми действиями, которые предусмотрены в подпрограмме. Описание подпрограммы помещается в разделе описаний (до начала исполняемых операторов).

Не вдаваясь в дальнейшие подробности, попробуем составить собственную процедуру, чтобы пояснить сказанное. Пусть в этой процедуре преобразуется некоторая символьная строка таким образом, чтобы все строчные буквы заменялись соответствующими прописными. В Delphi имеется стандартная функция `UpCase` (см. главу 6), которая выполняет аналогичные действия над одиночным символом. Наша процедура (назовем ее `UpString`) будет преобразовывать сразу все символы строки, причем сделаем ее пригодной не только для латинских букв, но и для букв русского алфавита¹.

Разработку программы проведем в два этапа. Сначала сконструируем основную (вызывающую) часть программы. Ее действия очень просты: она должна получить входную строку из компонента `edInput`, преобразовать ее с помощью процедуры `UpString` в выходную строку и поместить результат в `lbOutput.Text`. Эти действия нетрудно запрограммировать в обработчике `bbRunClick`:

```
procedure TfmExample.bbRunClick(Sender: TObject);
procedure UpString(stInp: String; var stOut: String);
  { Эта процедура преобразует все буквы строки stInp в
    прописные и помещает результат в выходную строку stOut }
begin
  stOut := stInp;
end;      // UpString
var
  s1, s2: String;
begin
  s1 := edInput.Text;      // Получаем исходную строку
```

¹ В Delphi имеется стандартная функция `AnsiUpperCase`, реализующая те же действия.


```

mmOutput.Lines.Add(S1);    // Выводим ее
UpString(S1,S2);           // Преобразуем
mmOutput.Lines.Add(S2);    // Выводим результат
edInput.Text := '';
edInput.SetFocus;
end;
```

В этой программе используется замещение процедуры UpString так называемой «заглушкой», то есть процедурой, в которой на самом деле не осуществляется нужных нам действий (однако она синтаксически абсолютно правильна, и при желании ее можно запустить), а выходная строка просто копирует входную. Заглушка понадобилась нам по двум причинам. Во-первых, приведенная программа очень проста, в ней отсутствует детальная реализация процедуры, и это позволяет наглядно проиллюстрировать механизм ее описания. Во-вторых, на ее примере мы знакомимся с универсальным методом конструирования сложных программ, получившим название *нисходящего программирования*. В соответствии с этим методом создание программы начинается «сверху», то есть с разработки самого главного (генерального) алгоритма. На верхнем уровне обычно еще не ясны детали реализации той или иной части программы, поэтому эти части следует заменить временными заглушками. Желательно, чтобы временный вариант программы был синтаксически правильным, тогда можно его откомпилировать и убедиться в отсутствии в нем синтаксических ошибок. Такой прогон даст определенную уверенность перед разработкой и реализацией алгоритмов нижнего уровня, то есть перед заменой заглушек реально работающими процедурами. Если реализуемый в заглушке алгоритм достаточно сложен, его вновь структурируют, выделяя главный алгоритм и применяя новые заглушки, и т. д. Процесс продолжается «вниз» до тех пор, пока не будет создан полностью работоспособный вариант программы.

Как видим, описание процедуры начинается зарезервированным словом **procedure**, за которым следуют имя процедуры и список формальных параметров. Список параметров заключается в круглые скобки и содержит перечень параметров с указанием их типа. Заметим, что перед параметром stOut, с помощью которого в вызывающую программу возвращается результат преобразования, стоит зарезервированное слово **var**. Именно таким способом компилятору указываются те параметры, в которых процедура возвращает вызвавшей ее программе результат своей работы (подробнее см. главу 10). Зарезервированное слово **procedure**, имя процедуры и список ее параметров образуют *заголовок процедуры*. За заголовком следует *тело процедуры*, содержащее новый раздел описаний (этот раздел пока еще пуст) и раздел исполняемых операторов (оператор stOut := stInp).

Приступим к разработке алгоритма процедуры. Для этого учтем, что в соответствии с принятой в Windows кодировкой символов (так называемой ANSI-кодировкой) символы кириллицы (русского алфавита) располагаются сплошным массивом от «А» до «я», за исключением «ё» и «Ё», которые предшествуют «А». Кроме того, символ «я» имеет внутренний код 255, то есть максимально возможное значение одного байта (для каждого символа строки выделяется по одному байту).

Вот возможный вариант процедуры:

```

Procedure UpString(stInp: String; var stOut: String);
```

```

{ Эта процедура преобразует все буквы строки stInp
  в прописные и помещает результат в выходную строку stOut }
var
  k: Integer;                // Параметр цикла
begin
  stOut := stInp;
  // Цикл побуквенного преобразования
  for k := 1 to Length(stOut) {Length - длина строки} do
  begin
    stOut[k] := UpCase(stOut[k]); // Преобразуем латиницу
    if stOut[k] >= 'a' then       // Строчная буква кириллицы?
      stOut[k] :=                // Да. Преобразуем ее
        Chr(ord('A') + ord(stOut[k]) - ord('a'));
    if stOut[k]='ё' then
      stOut[k] := 'Ё';           // Преобразуем ё в Ё
  end;
end; // UpString

```

В процедуре вначале исходная строка копируется в выходную строку с помощью оператора:

```
stOut := stInp;
```

Затем реализуется циклический перебор всех символов выходной строки для преобразования букв. Для доступа к отдельным символам строки используется замечательное свойство типа данных **String**, позволяющее рассматривать строку как набор (массив) символов. Первый символ этого набора имеет индекс 1, второй — 2 и т. д. Индекс указывается сразу за именем строки в квадратных скобках. Таким образом, `stOut[i]` — это *i*-й символ строки `stOut`. Перебор символов реализован в виде счетного цикла от 1 до длины входной строки `stOut` (эта длина получается с помощью стандартной функции `Length`), в ходе которого каждый символ сначала преобразуется функцией `UpCase` (эта функция возвращает прописную латинскую букву для строчной буквы, передаваемой ей в параметре вызова; функция не работает с кириллицей). Затем осуществляется проверка принадлежности буквы к русской строчной (оператор `if stOut[k] >= 'a' then...`; где *a* — русская буква) и осуществляется необходимая коррекция ее внутреннего кода. При коррекции используются отмеченные выше свойства непрерывности и монотонности массивов кодов, то есть за «А» идет «Б», за «Я» — «а», за «а» — «б» и т. д. Стандартная функция `ord` возвращает внутренний код символа, который передается ей в качестве параметра обращения, а функция `Chr` преобразует код в символ. Поскольку буква «ё» стоит особняком, ее преобразование осуществляется отдельным оператором.

Рассмотрим иной способ реализации той же программы — оформим алгоритм преобразования в виде *функции*:

```

procedure TfmExample.bbRunClick(Sender: TObject);
function UpString(stInp: String): String;
{ Эта функция преобразует все буквы строки stInp в пропис-
  ные и помещает результат в выходную строку Result }

```

```

var
  k: Integer;           // Параметр цикла
begin
  Result := stInp;
  // Цикл побуквенного преобразования
  for k := 1 to Length(Result) do
  begin
    Result[k] := UpCase(Result[k]); // Преобразуем латиницу
    if Result[k] >= 'a' then // Строчная буква кириллицы?
      Result[k] := // Да: преобразуем ее
        Chr(ord('A') + ord(Result[k]) - ord('a'));
    if Result[k]='ё' then
      Result[k] := 'Ё'; // Преобразуем ё в Ё
  end;
end; // UpString
begin
  mmOutput.Lines.Add(edInput.Text);
  { В следующем операторе исходная строка edInput.Text
  преобразуется и помещается в многострочное поле: }
  mmOutput.Lines.Add(UpString(edInput.Text));
  edInput.Text := '';
  edInput.SetFocus;
end;

```

При описании функции необходимо за списком используемых для ее вызова параметров указать тип возвращаемого ею результата. Именно поэтому за закрывающей круглой скобкой в заголовке функции стоит двоеточие и тип **String**. В теле любой функции определена стандартная переменная **Result**, которая трактуется как результат, возвращаемый функцией. С помощью следующего оператора мы сначала присвоили результату входную строку, а затем в цикле осуществили перебор символов и их коррекцию:

```
Result := stInp;
```

Единственный показанный ниже оператор заменил сразу три первых оператора в предыдущей реализации программы:

```
lbOutput.Caption := UpString(edInput.Text);
```

Правила кодирования программ

подавляющая часть кода программы располагается в многочисленных подпрограммах, поэтому ниже приводятся правила и рекомендации, относящиеся прежде всего к подпрограммам. Особенности кодирования модулей, библиотек, пакетов и собственно программ рассматриваются в соответствующих разделах книги.

Любая подпрограмма имеет заголовок, раздел объявлений и раздел операторов:

```

procedure MyProcedure; // Заголовок
// Раздел объявлений

```

```
begin
```

```
// Раздел операторов
```

```
end;
```

В разделе объявлений описываются переменные, константы, подпрограммы и метки, которые будут использованы в разделе операторов.

Описания группируются в секции, открываемые зарезервированными словами **var** (секция переменных), **const** (секция констант), **label** (секция меток). Порядок следования секций и их количество не имеют значения.

Описание переменной заключается в указании ее типа, описание константы — в указании ее значения; метка описывается простым упоминанием ее в секции **label**:

```
var
```

```
  I: Integer;
```

```
  a, b: Real;
```

```
  List: TStringList;
```

```
const
```

```
  s = 'Delphi';
```

```
  Max = 1234567;
```

```
label
```

```
  11, 12, 13;
```

В одной строке может размещаться произвольное количество описаний. Однотипные переменные можно группировать в списки (см. описание переменных а и b в предыдущем примере).

При описании глобальных переменных (то есть переменных, объявленных вне тела подпрограммы) их можно инициализировать, указав после названия типа знак равенства и начальное значение:

```
var
```

```
  Name: String = 'Delphi';
```

Локальные (то есть объявляемые в теле подпрограммы) переменные таким способом инициализировать нельзя.

В программе разрешается использовать типизированные константы:

```
const
```

```
  Name: String = 'Delphi';
```

В отличие от нетипизированных, типизированные константы можно изменять в ходе работы программы, то есть они ведут себя как инициализированные переменные.

ВНИМАНИЕ

Изменять значения типизированных констант можно во всех версиях Delphi, однако в версии 6 и 7 это становится возможным только при установленном флажке Assignable typed constants на вкладке Compiler окна Options (открывается командой Project ► Options).

Операторы располагаются в одной или нескольких строках и завершаются символом точки с запятой. Если оператор занимает несколько строк, он должен закан-

чиваться на каждой строке в том месте, где можно вставить пробел. Порядок следования операторов определяет порядок их исполнения, если не учитывать операторы переходов, которые нарушают этот порядок. В одной строке может размещаться несколько операторов. В этом случае они выполняются слева направо.

В любом месте, где можно вставить пробел, можно разместить комментарий. В Delphi в качестве ограничителей комментария используются символы {...}, (*...*) и //:

```
{ Это комментарий }
(* Это тоже комментарий *)
// Все символы до конца текущей строки образуют комментарий
```

Ограничители (**) и {} не могут пересекаться, но могут быть вложенными:

```
(*...{...*}...) // Так нельзя
(*...{...}*...) // Так можно
```

Если комментарий ограничивается символами {\$...} или (*\$...*), такой комментарий считается директивой компилятора. С его помощью программист может динамически изменять настройку компилятора и осуществлять условную компиляцию. Подробнее об использовании директив компилятора см. справочную службу Delphi (раздел Compiler Directives).

Излагаемые ниже советы «выстраданы» мною за 25 лет работы с языком Pascal. Однако если вы думаете, что они носят сугубо личный характер, вы глубоко ошибаетесь: посмотрите исходные тексты реализации любого модуля Delphi и вы увидите их на практике!

- ❖ Не размещайте на одной строке больше одного оператора. Этот совет связан с тем, что при пошаговой трассировке программ (а это — основной способ отладки), компилятор останавливается перед выполнением очередной строки кода. Если на строке размещены несколько операторов, вы не сможете проследить работу каждого оператора в этой строке отдельно от других.
- ❖ Выделяйте тела составных, условных, циклических операторов отступами. Если вы не будете этого делать, вам будет трудно определить, какому слову **begin** соответствует то или иное слово **end**. Delphi провоцирует широкое использование составных операторов, причем часто они бывают вложенными. Если пренебречь этим и двумя следующими советами, вы, возможно, потратите много времени на устранение ошибки, вызывающей сообщение **" ; " expected but " . " found** (не хватает **end**) или **Declaration expected but identifier found** (избыток **end**).
- ❖ Во вложенных составных операторах старайтесь помечать замыкающие слова **end** небольшими комментариями:

```
if a>b then
begin
...
if not IsNew then
begin
...
end; // if not IsNew
```

```
...
end; // if a>b
```

- Структурируйте алгоритм! Если тело вашей подпрограммы содержит более 20–30 строк, проанализируйте алгоритм и относительно самостоятельные его части вынесите во вложенные подпрограммы или реализуйте с помощью вспомогательных классов. Помните, что уже через 2–3 месяца вы будете с трудом припоминать детали реализации подпрограммы, насчитывающей сотни строк кода.
- Не скупитесь на комментарии. Небольшие затраты времени на вставку комментария *по ходу составления программы* с лихвой окупятся, когда вам понадобится ее модернизировать. Не откладывайте вставку комментария «на потом» — у меня, например, никогда на это «потом» не хватает времени.
- Старайтесь использовать осмысленные идентификаторы. К сожалению, для этого вам понадобится хотя бы начальное знание английского языка, в крайнем случае можно русскоязычные идентификаторы записывать латинскими буквами.

В заключение приведу небольшой фрагмент исходного кода модуля `buttons.pas` (находится в папке `Source\VCL` каталога размещения Delphi), иллюстрирующий применение перечисленных правил на практике.

```
with Canvas do
begin
  if NewStyle then
  begin
    Brush.Color := clBtnFace;
    Brush.Style := bsSolid;
    DC := Canvas.Handle;    { Reduce calls to GetHandle }

    if IsDown then
    begin { DrawEdge is faster than Polyline }
      DrawEdge(DC, R, BDR_SUNKENINNER, BF_TOPLEFT);
      { black }
      DrawEdge(DC, R, BDR_SUNKENOUTER, BF_BOTTOMRIGHT);
```

Часть II

Язык Delphi

В этой части книги описывается основной инструмент Delphi — язык программирования Delphi (в ранних версиях язык называется Object Pascal).

Глава 5

Элементы языка

Элементы языка кратко рассмотрены в главе 4. В этой главе приводятся более полные сведения.

Алфавит

Алфавит языка Delphi включает буквы, цифры, шестнадцатеричные цифры, специальные символы, пробелы и зарезервированные слова.

Буквы — это буквы латинского алфавита от а до z и от А до Z, а также знак подчеркивания (_). В языке нет различия между прописными и строчными буквами алфавита, если только они не входят в символьные и строковые выражения.

Цифры — арабские цифры от 0 до 9.

Каждая *шестнадцатеричная цифра* имеет значение от 0 до 15. Первые 10 значений обозначаются арабскими цифрами 0...9, остальные шесть — латинскими буквами А...F или а...f.

Специальные символы Delphi:

+ − ∗ / = , ' . : ; < > [] () { } ^ @ \$ #

К специальным символам относятся также следующие *пары* символов:

<> <= >= := (∗ ∗) (. .) //

В программе эти пары символов нельзя разделять пробелами, если они используются как знаки операций отношения или ограничители комментария. Символы (. и .) могут употребляться соответственно вместо [и].

Особое место в алфавите языка занимают *пробельные символы*, к которым относятся любые символы в диапазоне кодов от 0 до 32 (включая собственно пробел, а также символ табуляции). Эти символы рассматриваются как ограничители идентификаторов, констант, чисел, зарезервированных слов. Несколько следующих друг за другом пробелов считаются одним пробельным символом (последнее не относится к строковым константам).

В Delphi имеются следующие зарезервированные слова:

and	exports	mod	shr
array	file	nil	string

as	finalization	not	then
asm	finally	object	threadvar
begin	for	of	to
case	function	or	try
class	goto	out	type
const	if	packed	unit
constructor	implementation	procedure	until
destructor	in	program	uses
dispinterface	inherited	property	var
div	initialization	raise	while
do	inline	record	with
downto	interface	repeat	xor
else	is	resourcestring	
end	label	set	
except	library	shl	

Зарезервированные слова не могут использоваться в качестве идентификаторов.

Стандартные директивы первоначально связаны с некоторыми стандартными объявлениями в программе. К ним относятся:

absolute	dynamic	name	public	safecall
abstract	export	near	published	stdcall
assembler	external	nodefault	read	stored
automated	far	override	readonly	virtual
cdecl	forward	package	register	write
contains	implements	pascal	reintroduce	writelnonly
default	index	private	requires	
dispid	message	protected	resident	

Как и зарезервированные слова, стандартные директивы в окне кода Delphi выделяются жирным шрифтом, тем не менее вы можете переопределить любую стандартную директиву, то есть объявить одноименный идентификатор.

Слова **private**, **protected**, **public**, **published** и **automated** считаются зарезервированными внутри объявления класса и стандартными директивами — вне объявления.

Идентификаторы

Идентификаторы в Delphi — это имена констант, переменных, меток, типов, объектов, классов, свойств, процедур, функций, модулей, программ и полей в записях. Идентификаторы могут иметь произвольную длину.

Идентификатор всегда начинается буквой, за которой могут следовать буквы и цифры. Напомню, что буквой считается также символ подчеркивания, поэтому идентификатор может начинаться этим символом и даже состоять только из одного или нескольких символов подчеркивания. Пробелы и специальные символы алфавита не могут входить в идентификатор.

Примеры правильных идентификаторов:

```
a
MyProgramIsBestProgram
external
ALPHA
date_27_sep_39
_beta
```

Примеры неправильных идентификаторов:

```
1Program    // начинается цифрой
block#1     // содержит специальный символ
My Prog     // содержит пробел
mod         // зарезервированное слово
```

Константы

В качестве констант в Delphi могут использоваться целые, вещественные и шестнадцатеричные числа, логические константы, символы, строки символов, конструкторы множеств и признак неопределенного указателя **NIL**.

Целые числа записываются со знаком или без него по обычным правилам и могут иметь значение в диапазоне от -2^{63} до $+2^{63}-1$. Следует учесть, что если целочисленная константа выходит за указанные границы, компилятор генерирует сообщение об ошибке. Такие константы должны записываться с десятичной точкой, то есть определяться как вещественные числа.

Вещественные числа записываются со знаком или без него с использованием десятичной точки и/или экспоненциальной части. Экспоненциальная часть начинается символом *e* или *E*, за которым могут следовать знаки «+» или «-» и десятичный порядок. Символ *e* (*E*) означает десятичный порядок и имеет смысл «умножить на 10 в степени». Например,

3.14E5 — 3,14 умножить на 10 в степени 5;

-17e-2 — минус 17 умножить на 10 в степени минус 2.

Если в записи вещественного числа присутствует десятичная точка, перед точкой и за ней должно быть хотя бы по одной цифре. Если используется символ экспоненциальной части *e* (*E*), за ним должна следовать хотя бы одна цифра десятичного порядка.

Шестнадцатеричное число состоит из шестнадцатеричных цифр, которым предшествует знак доллара \$ (код символа — 36). Диапазон шестнадцатеричных чисел — от \$FFFFFFFFFFFFFFFF до \$FFFFFFFFFFFFFFFF.

Логическая константа — это либо слово *False* (ложь), либо слово *True* (истина).

Символьная константа — это любой символ ПК, заключенный в апострофы:

'z' — символ «z»;

'ф' — символ «Ф».

Если необходимо записать собственно символ апострофа, он удваивается:

''' — символ апострофа (');

'That' 's string' — строка с символом апострофа.

Допускается использование записи символа путем указания его внутреннего кода, которому предшествует символ # (код 35), например:

```
#97 — символ «а»;
#90 — символ «Z»;
#39 — символ «'»;
#13 — символ «CR».
```

Строковая константа — любая последовательность символов (кроме символов CR — возврат каретки), заключенная в апострофы.

Строка символов может быть пустой, то есть не иметь никаких символов в обрамляющих ее апострофах. Строку можно составлять из кодов нужных символов с предшествующими каждому коду символами #, например, следующая строка эквивалентна строке Symbol:

```
#83#121#109#98#11#108
```

Наконец, в строке можно чередовать части, записанные в обрамляющих апострофах, с частями, записанными кодами. Таким способом можно вставлять в строки любые управляющие символы, в том числе и символы CR (код 13), например:

```
#7'Ошибка !'#13'Нажмите любую клавишу ...'#7
```

Конструктор множества — список элементов множества, обрамленный квадратными скобками, например:

```
[1, 2, 4..7, 12]
[blue, red]
[]
[true]
```

В Delphi разрешается в объявлении констант использовать произвольные *выражения*, операндами которых могут быть ранее объявленные нетипизированные константы, имена типов и объектов, а также следующие функции от них:

abs	length	ord	round	swap
chr	low	pred	sizeof	trunc
hi	odd	ptr	succ	

Например:

```
const
  MaxReal = MaxInt div SizeOf(real);
  NumChars = ord('Z') - ord('a') + 1;
  Ln10     = 2.302585092994;
  Ln10R    = 1 / Ln10;
```

Выражения

Основными элементами, из которых конструируется исполняемая часть программы, являются константы, переменные и обращения к функциям. Каждый из этих элементов характеризуется своим значением и принадлежит к какому-либо типу данных. С помощью знаков операций и скобок из них можно составлять *выражения*, которые фактически представляют собой правила получения новых значений.

Частным случаем выражения может быть просто одиночный элемент, то есть константа, переменная или обращение к функции. Значение такого выражения имеет, естественно, тот же тип, что и сам элемент. В общем случае выражение состоит из нескольких элементов (операндов) и знаков операций, а тип его значения определяется типом операндов и видом примененных к ним операций.

Примеры выражений:

```
y
21
(a + b) * c
sin(t)
a > 2
not Flag and (a = b)
NIL
[1, 3..7] * set1
```

Операции

В Delphi определены следующие операции:

- унарные (**not**, **@**);
- мультипликативные (*****, **/**, **div**, **mod**, **and**, **shl**, **shr**);
- аддитивные (**+**, **-**, **or**, **xor**);
- отношения (**=**, **<>**, **<**, **>**, **<=**, **>=**, **in**).

Приоритет операций убывает в указанном порядке, то есть высшим приоритетом обладают унарные операции, низшим — операции отношения. Порядок выполнения нескольких операций равного приоритета устанавливается компилятором из условия оптимизации кода программы и необязательно слева направо. При исчислении логических выражений операции равного приоритета всегда вычисляются слева направо, причем будут вычисляться все или только достаточные операции в зависимости от установленного флажка **Complete Boolean eval** на вкладке **Compiler** диалогового окна **Options** (вызывается командой **Project ▶ Options**): при установленном флажке вычисляются все операции отношения, при снятом — только те, что необходимы для однозначного определения результата исчисления.

Правила использования операций с операндами различного типа иллюстрирует табл. 5.1.

Таблица 5.1. Правила использования операций

Операция	Действие	Тип операндов	Тип результата
not	Отрицание	Логический	Логический
not	То же	Любой целый	Тип операнда
@	Адрес	Любой	Указатель
*	Умножение	Любой целый	Наименьший целый
*	То же	Любой вещественный	Extended
*	Пересечение множеств	Множественный	Множественный

продолжение ➤

Таблица 5.1 (продолжение)

Операция	Действие	Тип операндов	Тип результата
/	Деление	Любой вещественный	Extended
div	Целочисленное деление	Любой целый	Наименьший целый
mod	Остаток от деления	То же	То же
and	Логическое И	Логический	Логический
and	То же	Любой целый	Наименьший целый
shl	Левый сдвиг	То же	То же
shr	Правый сдвиг	То же	То же
+	Сложение	То же	То же
+	То же	Любой вещественный	Extended
+	Объединение множеств	Множественный	Множественный
+	Сцепление строк	Строковый	Строковый
–	Вычитание	Любой целый	Наименьший целый
–	То же	Любой вещественный	Extended
or	Логическое ИЛИ	Логический	Логический
or	То же	Любой целый	Наименьший целый
=	Равно	Любой простой или строковый	Логический
<>	Не равно	То же	То же
<	Меньше	Логический	Логический
<=	Меньше или равно	То же	То же
>	Больше	То же	То же
>=	Больше или равно	То же	То же

Унарная операция @ применяется к операнду любого типа и возвращает результат типа `Pointer` (см. главу 8), в котором содержится адрес операнда. Если операция @ применяется к процедуре, функции или методу в объекте, ее результатом будет адрес точки входа в эту процедуру (функцию, метод). Этот адрес можно использовать только в подпрограмме, написанной на ассемблере, или во встраиваемых фрагментах (задаются ключевым словом **inline**).

В Delphi определены следующие логические операции:

- **not** — логическое НЕ;
- **and** — логическое И;
- **or** — логическое ИЛИ;
- **xor** — исключающее ИЛИ.

Логические операции применимы к операндам целого и логического типов. Если операнды — целые числа, то результат логической операции есть тоже целое число, биты которого (двоичные разряды) формируются из битов операндов по правилам, указанным в табл. 5.2.

Таблица 5.2. Логические операции (поразрядные) над данными целого типа

Операнд 1	Операнд 2	not	and	or	xor
1	Нет	0	Нет	Нет	Нет
0	Нет	1	Нет	Нет	Нет
0	0	Нет	0	0	0
0	1	Нет	0	1	1
1	0	Нет	0	1	1
1	1	Нет	1	1	0

К логическим в Delphi обычно относятся еще две сдвиговые операции над целыми числами:

- ✱ **i shl j** — сдвиг содержимого **i** на **j** разрядов влево; освободившиеся младшие разряды заполняются нулями;
- ✱ **i shr j** — сдвиг содержимого **i** на **j** разрядов вправо; освободившиеся старшие разряды заполняются нулями.

В этих операциях **i** и **j** — выражения любого целого типа.

Логические операции над логическими данными дают результат логического типа по правилам, указанным в табл. 5.3.

Таблица 5.3. Логические операции над данными логического типа

Операнд 1	Операнд 2	not	and	or	xor
True	Нет	False	Нет	Нет	Нет
False	Нет	True	Нет	Нет	Нет
False	False	Нет	False	False	False
False	True	Нет	False	True	True
True	False	Нет	False	True	True
True	True	Нет	True	True	False

Операция отношения **in** применяется к двум операндам. Первым (левым) операндом должно быть выражение любого порядкового типа, вторым — множество, состоящее из элементов того же типа, или идентификатор множественного типа. Результат операции равен True, если левый операнд принадлежит множеству.

Глава 6

Типы данных

Любые данные, то есть константы, переменные, свойства, значения функций или выражений в Delphi характеризуются своими типами. *Тип* определяет множество допустимых значений, которые может иметь тот или иной объект, а также множество допустимых операций, которые применимы к нему. Кроме того, тип определяет также и формат внутреннего представления данных в памяти ПК.

Delphi характеризуется разветвленной структурой типов данных (рис. 6.1). В языке предусмотрен механизм создания новых типов, благодаря чему общее количество используемых в программе типов может быть сколь угодно большим.

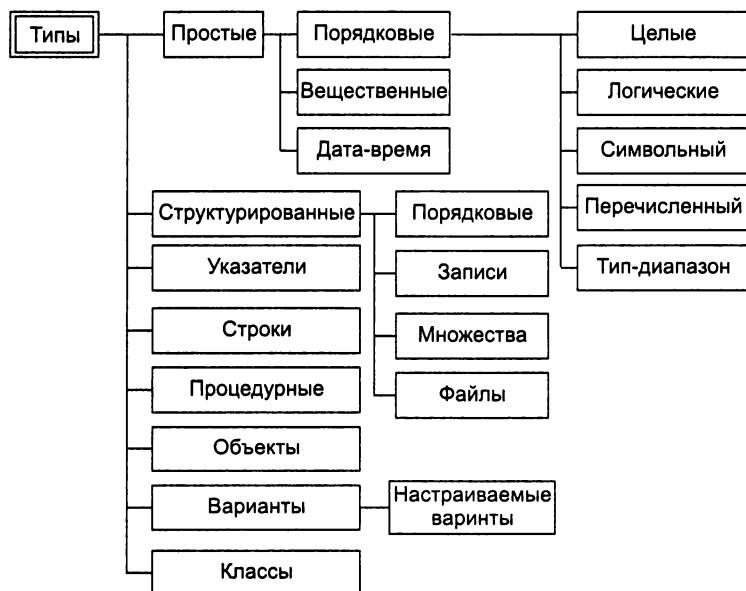


Рис. 6.1. Типы данных в Delphi

В этой и четырех последующих главах приводится подробное описание всех типов.

Простые типы

К простым типам относятся порядковые и вещественные типы, а также тип дата-время.

Порядковые типы отличаются тем, что каждый из них имеет конечное количество возможных значений. Эти значения можно определенным образом упорядочить (отсюда — название типов) и, следовательно, с каждым из них можно сопоставить некоторое целое число — порядковый номер значения.

Вещественные типы, строго говоря, тоже имеют конечное количество значений, которое определяется форматом внутреннего представления вещественного числа. Однако это количество настолько велико, что сопоставить с каждым из них целое число (его номер) не представляется возможным.

Тип *дата-время* предназначен для хранения даты и времени. Фактически для этих целей он использует вещественный формат.

Порядковые типы

К порядковым типам относятся (см. рис. 6.1) целые, логические, символьный, перечисленный и тип-диапазон. К любому из них применима функция $\text{Ord}(X)$, которая возвращает порядковый номер значения выражения X .

Для целых типов функция $\text{Ord}(X)$ возвращает само значение X , то есть $\text{Ord}(X) = X$ для значения X , принадлежащего любому целому типу. Применение $\text{Ord}(X)$ к логическому, символьному и перечисленному типам дает положительное целое число в диапазоне от 0 до 1 (логический тип), от 0 до 255 (символьный), от 0 до 65535 (перечисленный). Тип-диапазон сохраняет все свойства базового порядкового типа, поэтому результат применения к нему функции $\text{Ord}(X)$ зависит от свойств этого типа.

К порядковым типам можно также применять следующие функции:

- $\text{Pred}(X)$ — возвращает предыдущее значение порядкового типа (значение, которое соответствует порядковому номеру $\text{Ord}(X) - 1$), то есть:

$$\text{Ord}(\text{Pred}(X)) = \text{Ord}(X) - 1$$

- $\text{Succ}(X)$ — возвращает следующее значение порядкового типа, которое соответствует порядковому номеру $\text{Ord}(X) + 1$, то есть:

$$\text{Ord}(\text{Succ}(X)) = \text{Ord}(X) + 1$$

Например, пусть в программе определена переменная:

```
var
  c: Char;
begin
  c:= '5';
end;
```

Тогда функция $\text{Pred}(C)$ вернет символ '4', а функция $\text{Succ}(C)$ — символ '6'.

Если представить себе любой порядковый тип как упорядоченное множество значений, возрастающих слева направо и занимающих на числовой оси некото-

рый отрезок, то функция $\text{Pred}(X)$ не определена для левого, а $\text{Succ}(X)$ — для правого конца этого отрезка.

Целые типы

Диапазон возможных значений целых типов зависит от их внутреннего представления, которое может занимать один, два, четыре или восемь байтов. В табл. 6.1 приводятся названия целых типов, длина их внутреннего представления в байтах и диапазон возможных значений.

Таблица 6.1. Целые типы

Название	Длина, байтов	Диапазон значений
Byte	1	0...255
ShortInt	1	-128...+127
SmallInt	2	-32 768...+32 767
Word	2	0...65 535
Integer	4	-2 147 483 648...+2 147 483 647
LongInt	4	-2 147 483 648...+2 147 483 647
LongWord	4	0...4 294 967 295
Int64	8	-9×1018...+9×1018
Cardinal	4	0... 2 147 483 647

При использовании процедур и функций с целочисленными параметрами следует руководствоваться «вложенностью» типов, то есть везде, где может использоваться тип `Word`, допускается использование типа `Byte` (но не наоборот), в `LongInt` «входит» `SmallInt`, который, в свою очередь, включает в себя `ShortInt`.

Перечень процедур и функций, применимых к целочисленным типам, приведен в табл. 6.2. Буквами `b`, `s`, `w`, `i`, `l` обозначены выражения соответственно типа `Byte`, `ShortInt`, `Word`, `Integer` и `LongInt`, `x` — выражение любого из этих типов; буквы `vb`, `vs`, `vw`, `vi`, `vl`, `vx` обозначают переменные соответствующих типов. В квадратных скобках указывается необязательный параметр.

Таблица 6.2. Стандартные процедуры и функции, применимые к целым типам

Обращение	Тип результата	Действие
<code>abs(x)</code>	<code>x</code>	Возвращает модуль <code>x</code>
<code>chr(b)</code>	<code>Char</code>	Возвращает символ по его коду
<code>dec(vx[, i])</code>	—	Уменьшает значение <code>vx</code> на <code>i</code> , а при отсутствии <code>i</code> — на 1
<code>inc(vx[, i])</code>	—	Увеличивает значение <code>vx</code> на <code>i</code> , а при отсутствии <code>i</code> — на 1
<code>Hi(w)</code>	<code>Byte</code>	Возвращает старший байт аргумента
<code>Hi(I)</code>	То же	Возвращает третий по счету байт
<code>Lo(i)</code>	То же	Возвращает младший байт аргумента

Обращение	Тип результата	Действие
Lo(w)	То же	То же
odd(l)	Boolean	Возвращает True, если аргумент — нечетное число
Random(w)	Как у параметра	Возвращает псевдослучайное число, равномерно распределенное в диапазоне 0... (w-1)
sqr(x)	x	Возвращает квадрат аргумента
swap(i)	Integer	Меняет местами байты в слове
swap(w)	Word	То же

При действиях с целыми числами тип результата будет соответствовать типу операндов, а если операнды относятся к различным целым типам, — общему типу, который включает в себя оба операнда. Например, при действиях с ShortInt и Word общим будет тип Integer. При стандартной настройке компилятор Delphi не вырабатывает код, осуществляющий контроль за возможной проверкой выхода значения из допустимого диапазона, что может привести к недоразумениям. Например, при прогоне следующей программы на экране появится значение 0:

```
procedure TfmExample.bbRunClick(Sender: TObject);
var
    k: Word;
begin
    k := 65535;                // Максимальное значение
                                // типа Word
    k := k+1;                  // По правилам математики
                                // k=65536
    lbOutput.Caption := IntToStr(k); // На самом деле k=0!
end;
```

Если установить флажок Range checking на вкладке Compiler диалогового окна Options (открывается командой Project ► Options) и повторить компиляцию с помощью команды Project ► Build All, компилятор вставит в программу код проверки переполнения и при прогоне программы возникнет исключительная ситуация, которую при желании можно соответствующим образом обработать. Замечу, что если изменить программу так, как показано ниже, переполнения не произойдет, так как 32-разрядный компилятор версий Delphi автоматически преобразует операнды выражения k+1 к 4-байтным величинам:

```
k := 65535;                // Максимальное значение
                                // типа Word
lbOutput.Caption := IntToStr(k + 1); // Будет выведено
                                // 65536
```

При обращениях к функциям API Windows могут использоваться следующие целые типы, определенные в модуле Windows (табл. 6.3).

Таблица 6.3. Целые типы для связи с ядром Windows

Тип Windows	Базовый тип	Тип Windows	Базовый тип
DWORD	LongWord	UCHAR	Byte

продолжение ➤

Таблица 6.3 (продолжение)

Тип Windows	Базовый тип	Тип Windows	Базовый тип
LANGID	Word	UINT	LongWord
LCID	LongWord	ULONG	Cardinal
SHORT	SmallInt		

Логические типы

К логическим относятся типы `Boolean`, `ByteBool`, `Bool`, `WordBool` и `LongBool`. В стандартном языке Паскаль определен только тип `Boolean`, остальные логические типы введены в Delphi для совместимости с Windows: типы `Boolean` и `ByteBool` занимают по одному байту каждый, `Bool` и `WordBool` — по 2 байта, `LongBool` — 4 байта. Значениями логического типа может быть одна из предварительно объявленных констант `False` (ложь) или `True` (истина). Для них справедливы правила:

```
Ord(False) = 0
Ord(True) <> 0
Succ(False) = True
Pred(True) = False
```

Поскольку логический тип относится к порядковым типам, его можно использовать в операторе цикла счетного типа, например:

```
var
  l : Boolean;
begin
  for l := False to True do ...
```

Однако необходимо помнить, что в 32-разрядных версиях Delphi для `Boolean` значение `Ord(True) = +1`, в то время как для других типов (`Bool`, `WordBool` и т. д.) `Ord(True) = -1`, поэтому такого рода операторы следует использовать с осторожностью¹. Например, для версии Delphi 7 исполняемый оператор `ShowMessage('---')` в следующем цикле `for` не будет выполнен ни разу:

```
var
  L: Bool;
begin
  for L := False to True do
    ShowMessage('---');
end;
```

Если заменить тип параметра цикла `L` в предыдущем примере на `Boolean`, цикл будет работать, и сообщение дважды появится на экране.

Символьный тип

Значениями символьного типа является множество всех символов клавиатуры компьютера. Каждому символу приписывается целое число в диапазоне 0...255. Это

¹ Для Delphi версий 1 и 2 `ord(True) = +1` для любого логического типа.

число служит кодом внутреннего представления символа, его возвращает функция `ord`.

Для кодировки в Windows используется код ANSI (назван по имени American National Standard Institute — американский институт стандартизации, предложивший этот код). Первая половина символов ПК с кодами 0...127 представлена в табл. 6.4. Вторая половина символов с кодами 128...255 меняется для различных шрифтов. Стандартные Windows-шрифты Arial, Courier New, Times New Roman для представления символов кириллицы (без букв «ё» и «Ё») используют последние 64 кода (от 192 до 256): «А»...«Я» кодируются значениями 192...223, «а»...«я» — 224...255. Символы «Ё» и «ё» имеют соответственно коды 168 и 184.

Таблица 6.4. Кодировка символов в соответствии со стандартом ANSI

Код	Символ	Код	Символ	Код	Символ	Код	Символ
0	NUL	32	BL	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DEL	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y

продолжение ➤

Таблица 6.4 (продолжение)

Код	Символ	Код	Символ	Код	Символ	Код	Символ
26	SUB	58	:	90	z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	▯

Символы с кодами 0...31 относятся к служебным кодам. Если эти коды используются в тексте программы, они считаются пробелами.

К типу Char применимы операции отношения, а также встроенные функции:

- ✎ chr (B) — функция типа Char; преобразует выражение B типа Byte в символ и возвращает его своим значением;
- ✎ UpCase (CH) — функция типа Char; возвращает прописную букву, если CH — строчная латинская буква, в противном случае возвращает сам символ CH (для кириллицы возвращает исходный символ).

Перечисленный тип

Перечисленный тип задается перечислением тех значений, которые он может получать. Каждое значение именуется некоторым идентификатором и располагается в списке, обрамленном круглыми скобками, например:

```
type
  colors = (red, white, blue);
```

Применение перечисленных типов делает программы нагляднее. Если, например, в программе используются данные, связанные с месяцами года, то следующий фрагмент программы был бы, согласитесь, очень наглядным:

```
type
  ТипМесяц= (янв, фев, мар, апр, май, июн, июл, авг, сен, окт, ноя, дек) ;
var
  месяц : ТипМесяц;
begin
  ...
  if месяц = авг then
    lbOutput.Caption := 'Хорошо бы поехать к морю!';
  ...
end.
```

Увы! В Delphi нельзя использовать кириллицу в идентификаторах, поэтому мы вынуждены писать так:

```
type
  TypeMonth= (jan, feb, mar, may, jun, jul, aug, sep, oct, nov, dec) ;
var
  month: TypeMonth;
```

```

begin
    ...
    if month = aug then
        lbOutput.Caption := 'Хорошо бы поехать к морю!';
    ...
end.

```

Соответствие между значениями перечисленного типа и порядковыми номерами этих значений устанавливается очередностью перечисления: первое значение в списке получает порядковый номер 0, второе — 1 и т. д. Максимальная мощность перечисленного типа составляет 65 536 значений, поэтому фактически перечисленный тип задает некоторое подмножество целого типа `Word` и может рассматриваться как компактное объявление сразу группы целочисленных констант со значениями 0, 1 и т. д.

Использование перечисленных типов повышает надежность программ благодаря возможности контроля тех значений, которые получают соответствующие переменные. Пусть, например, заданы такие перечисленные типы:

```

type
    colors = (black, red, white);
    ordinal = (one, two, three);
    days    = (Monday, Tuesday, Wednesday);

```

С точки зрения мощности и внутреннего представления все три типа эквивалентны:

- `Ord(black)=0, ..., Ord(white)=2,`
- `Ord(one)=0, ..., Ord(three)=2,`
- `Ord(Monday)=0, ..., Ord(Wednesday)=2.`

Однако рассмотрим случай, когда определены следующие переменные:

```

var
    col : colors;
    num : ordinal;
    day : days;

```

Тогда перечисленные ниже операторы допустимы:

```

col := black;
num := Succ(two);
day := Pred(Tuesday);

```

В то время как следующие операторы недопустимы:

```

col := one;
day := black;

```

Как уже упоминалось, между значениями перечисленного типа и множеством целых чисел существует однозначное соответствие, задаваемое функцией `Ord(X)`. В Delphi допускается и обратное преобразование: любое выражение типа `Word` можно преобразовать в значение перечисленного типа, если только значение целочисленного выражения не превышает мощности этого типа. Такое преобразование достигается применением автоматически объявленной функции с именем `pe-`

речисленного типа. Например, для рассмотренного выше объявления типов эквивалентны следующие присваивания:

```
col := black;
col := colors(0);
```

Разумеется, представленное ниже присваивание будет недопустимым, так как перечисленному типу нельзя присвоить целое значение:

```
col := 0;
```

Переменные любого перечисленного типа можно объявлять без предварительного описания этого типа, например:

```
var
  col: (black, white, green);
```

Тип-диапазон

Тип-диапазон есть подмножество своего базового типа, в качестве которого может выступать любой порядковый тип, кроме типа-диапазона.

Тип-диапазон задается границами своих значений внутри базового типа:

```
<мин.знач.>..макс.знач.>
```

Здесь **<мин.знач.>** — минимальное значение типа-диапазона; **<макс.знач.>** — максимальное его значение.

Например:

```
type
  digit   = '0'..'9';
  dig2    = 48 .. 57;
```

Тип-диапазон необязательно описывать в разделе **type**, а можно указывать непосредственно при объявлении переменной, например:

```
var
  date : 1..31;
  month: 1..12;
  lchr : 'A'..'Z';.
```

При определении типа-диапазона нужно руководствоваться следующими правилами:

- два символа точки (..) рассматриваются как один символ, поэтому между ними недопустимы пробелы;
- левая граница диапазона не должна превышать его правую границу.

Тип-диапазон наследует все свойства своего базового типа, но с ограничениями, связанными с его меньшей мощностью. В частности, пусть определена переменная:

```
type
  days    = (mo,tu,we,th,fr,sa,su);
  Weekend = sa .. su;

var
  w : Weekend;
```

```

begin
  ...
  w := sa;
  ...
end;

```

Тогда функция `Ord (W)` вернет значение 5, в то время как `Pred (W)` приведет к ошибке.

В стандартную библиотеку Delphi включены две функции, поддерживающие работу с типами-диапазонами:

- ▣ `High (X)` — возвращает максимальное значение типа-диапазона, к которому принадлежит переменная `X`;
- ▣ `Low (X)` — возвращает минимальное значение типа-диапазона.

Вещественные типы

В отличие от порядковых типов, значения которых всегда сопоставляются с рядом целых чисел и, следовательно, представляются в ПК абсолютно точно, значения вещественных типов определяют произвольное число лишь с некоторой конечной точностью, зависящей от внутреннего формата вещественного числа (табл. 6.5).

Таблица 6.5. Вещественные типы

Длина, байтов	Название	Количество значащих цифр	Диапазон значений
8	Real	15...16	$5,0 \times 10^{-324} \dots 1,7 \times 10^{308}$
4	Single	6...8	$1,5 \times 10^{-45} \dots 3,4 \times 10^{38}$
8	Double	15...16	$5,0 \times 10^{-324} \dots 1,7 \times 10^{308}$
10	Extended	19...20	$3,4 \times 10^{-4951} \dots 1,1 \times 10^{4932}$
8	Comp	19...20	$-2^{63} \dots +2^{63} - 1$
8	Currency	19...20	$\pm 922\,337\,203\,685\,477,5807$

ПРИМЕЧАНИЕ В версиях Delphi 1...3 тип `Real` занимал 6 байт и имел диапазон значений от $2,9 \times 10^{-39}$ до $1,7 \times 10^{38}$. В версиях 4...7 этот тип эквивалентен типу `Double`. Если требуется (в целях совместимости) использовать 6-байтный тип `Real`, нужно указать директиву компилятора `{ $REALCOMPATIBILITY ON }`.

Как видно из табл. 6.5, вещественное число в Delphi занимает от 4 до 10 смежных байтов и имеет следующую структуру в памяти компьютера:

s	e	m
---	---	---

Здесь `s` — знаковый разряд числа; `e` — экспоненциальная часть; содержит двоичный порядок; `m` — мантисса числа.

Мантисса `m` имеет длину от 23 (для `Single`) до 63 (для `Extended`) двоичных разрядов, что и обеспечивает точность 6...8 для `Single` и 19...20 для `Extended`

десятичных цифр. Десятичная точка (запятая) подразумевается перед левым (старшим) разрядом мантиссы, но при действиях с числом ее положение сдвигается влево или вправо в соответствии с двоичным порядком числа, хранящимся в экспоненциальной части, поэтому действия над вещественными числами называют арифметикой с плавающей точкой (запятой).

Отметим, что арифметический сопроцессор всегда обрабатывает числа в формате `Extended`, а три других вещественных типа в этом случае получаются простым усечением результатов до нужных размеров и применяются в основном для экономии памяти.

Например, пусть «машинное эpsilon» (см. описание программы `Epsilon` в главе 4) вычисляется с помощью такой программы:

```
type
  RealType = Real;
var
  Epsilon : RealType;
begin
  Epsilon := 1;
  while 1+Epsilon/2 > 1 do
    Epsilon := Epsilon/2;
  lbOutput.Caption := FloatToStr(Epsilon)
end;
```

Тогда независимо от объявления типа `RealType` (он может быть `Single`, `Real`, `Double` или `Extended`) на печать будет выведен следующий результат:

```
1,08420217248550E-0019
```

Это соответствует типу `Extended`, поскольку все операнды вещественного выражения `1+Epsilon/2` в операторе `while` перед вычислением автоматически преобразуются к типу `Extended`. Чтобы получить правильный результат, программе необходимо изменить следующим образом:

```
type
  RealType = Real;
var
  Epsilon, Eps1 : RealType;
begin
  Epsilon := 1;
  repeat
    Epsilon := Epsilon/2;
    Eps1 := 1 + Epsilon
  until Eps1 = 1;
  lbOutput.Caption := FloatToStr(2*Epsilon)
end.
```

Особое положение в Delphi занимают типы `Comp` и `Currency`, которые трактуются как вещественные числа с дробными частями фиксированной длины: в `Comp` дробная часть имеет длину 0 разрядов, то есть просто отсутствует, в `Currency` длина дробной части составляет 4 десятичных разряда. Фактически, оба типа оп-

ределяют большое целое число со знаком, сохраняющее 19...20 значащих десятичных цифр (во внутреннем представлении они занимают 8 смежных байтов). В то же время в выражениях `Comp` и `Currency` полностью совместимы с любыми другими вещественными типами: над ними определены все вещественные операции, они могут использоваться как аргументы математических функций и т. д. Наиболее подходящей областью применения этих типов являются бухгалтерские расчеты.

Для работы с вещественными данными могут использоваться стандартные математические функции, представленные в табл. 6.6. В этой таблице `Real` означает любой вещественный тип, `Integer` — любой целый тип.

Таблица 6.6. Стандартные математические функции Delphi

Обращение	Тип параметра	Тип результата	Примечание
<code>abs(x)</code>	<code>Real</code> , <code>Integer</code>	Тип аргумента	Модуль аргумента
<code>ArcTan(x)</code>	<code>Real</code>	<code>Real</code>	Арктангенс (значение в радианах)
<code>cos(x)</code>	То же	То же	Косинус, угол в радианах
<code>exp(x)</code>	То же	То же	Экспонента
<code>frac(x)</code>	То же	То же	Дробная часть числа
<code>int(x)</code>	То же	То же	Целая часть числа
<code>ln(x)</code>	То же	То же	Логарифм натуральный
<code>pi</code>	—	То же	$\pi = 3.141592653...$
<code>Random</code>	—	То же	Псевдослучайное число, равномерно распределенное в диапазоне 0...[1]
<code>Random(x)</code>	<code>Integer</code>	<code>Integer</code>	Псевдослучайное целое число, равномерно распределенное в диапазоне 0... (x-1)
<code>Randomize</code>	—	—	Инициализация генератора псевдослучайных чисел
<code>sin(x)</code>	<code>Real</code>	<code>Real</code>	Синус, угол в радианах
<code>sqr(x)</code>	То же	То же	Квадрат аргумента
<code>sqrt(x)</code>	То же	То же	Корень квадратный

Указанная в таблице функция `Random` представляет собой генератор псевдослучайных чисел. Эта функция получает некоторое целое число, называемое базовым, изменяет его разряды по определенному алгоритму и выдает новое число как результат. Одновременно с этим новое число становится базовым при следующем обращении к функции и т. д. (Так как алгоритм процедуры не меняется в ходе ее работы, числа называются *псевдослучайными*.) В системном модуле `System`, который автоматически доступен любой программе, базовое число хранится в переменной с именем `RandSeed` и всегда имеет начальное значение 0. Это означает, что при последовательном обращении к функции `Random` в разных программах (или при нескольких прогонах одной программы) будет всегда выдана одна и та

же последовательность псевдослучайных чисел. Например, при следующем обработчике `bbRunClick` окно нашей учебной программы будет выглядеть так, как показано на рис. 6.2:

```
procedure TfmExample.bbRunClick(Sender: TObject);
var
    k: Integer;
    S,SS: String;
begin
    S := '';
    for k := 1 to 300 do
        begin
            SS := IntToStr(Random(1000));
            while Length(SS)<3 do
                SS := '0'+SS;
            S := S+SS+' ';
            if k mod 20=0 then
                begin
                    mmOutput.Lines.Add(S);
                    S := '';
                end
            end
        end;
end;
```

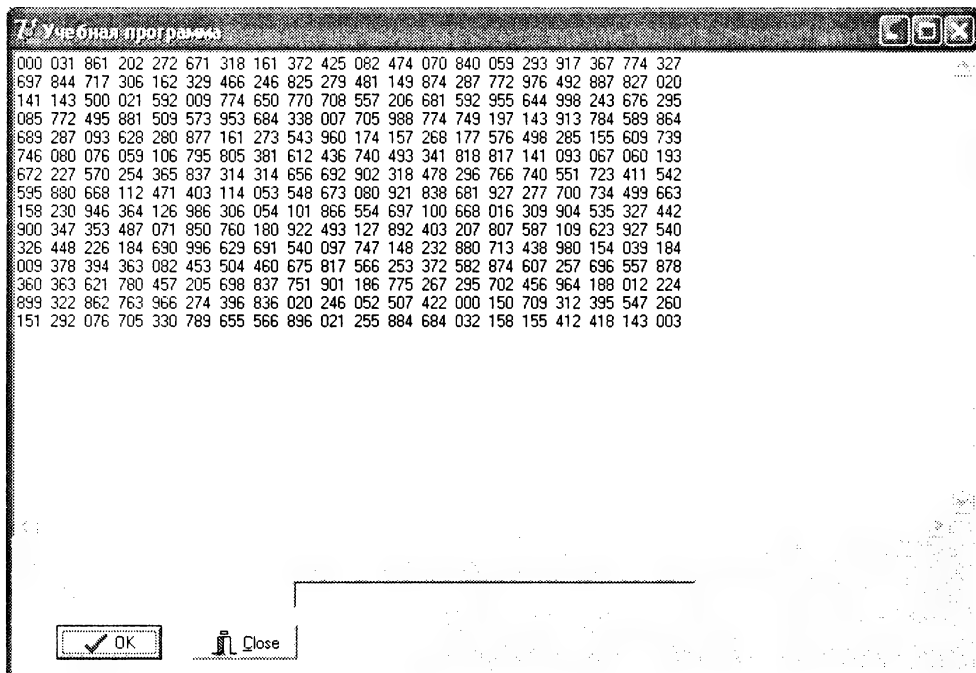


Рис. 6.2. Последовательность псевдослучайных чисел

В программе выводятся 300 псевдослучайных чисел, которые группируются в строки по 20 чисел и дополняются ведущими нулями, если число имеет меньше трех цифр. Если вы захотите повторить программу, вы получите точно такую же последовательность чисел, что и на рис. 6.2. С помощью процедуры `Randomize` в переменную `RandSeek` можно поместить численное значение системного времени, что приведет к генерации другой последовательности. Обращение к этой процедуре нужно сделать в самом начале обработчика.

В Delphi включен модуль `Match`, который существенно расширяет перечисленный в табл. 6.6 набор встроенных математических функций. Особенностью реализации содержащихся в нем более 70 функций и процедур является их оптимизация для работы с арифметическим сопроцессором класса `Pentium`, так что все они производят необходимые вычисления за рекордно малое время. Исходный текст модуля содержится в файле `Source\Rtl\Sys\Match.pas` каталога размещения Delphi. В приложении Б перечисляются некоторые подпрограммы модуля `Match`.

Тип дата-время

Тип дата-время определяется стандартным идентификатором `TDateTime` и предназначен для одновременного хранения и даты, и времени. Во внутреннем представлении он занимает 8 байт и, подобно типу `Currency`, представляет собой вещественное число с фиксированной дробной частью: в целой части числа хранится дата, в дробной — время. Дата определяется как количество суток, прошедших с 30 декабря 1899 г., а время — как часть суток, прошедших с 0 часов, так что значение 36444,837 соответствует дате 11.10.1999 и времени 20:05. Количество суток может быть и отрицательным, однако значения, меньшие -693 594 (соответствует дате 00.00.0000 от Рождества Христова), игнорируются функциями преобразования даты к строковому типу.

Над данными типа `TDateTime` определены те же операции, что и над вещественными числами, а в выражениях этого типа могут участвовать константы и переменные целого и вещественного типов. Для работы с датой и временем используются подпрограммы, некоторые из которых перечислены в табл. 6.7.

Таблица 6.7. Подпрограммы для работы с датой и временем

Подпрограмма	Описание
function Date: TDateTime;	Возвращает текущую дату
function DateToStr(D: TDateTime): String;	Преобразует дату в строку символов
function DateTimeToStr(D: TDateTime): String;	Преобразует дату и время в строку символов
function FormatDateTime (Format: String; Value: TDateTime): String;	Преобразует дату и время из параметра Value в строку символов в соответствии со спецификаторами параметра Format (см. пояснения в разделе «Строки»)
function Now: TDateTime;	Возвращает текущие дату и время
function Time: TDateTime;	Возвращает текущее время
function TimeToStr(T: TDateTime): String;	Преобразует время в строку

Полный перечень функций для работы с датой-временем приводится в приложении Б.

Поскольку тип `TDateTime` совместим с форматом вещественных чисел, можно без труда определить дату, отстоящую от заданной на сколько-то дней вперед или назад: для этого достаточно соответственно прибавить к заданной дате или отнять от нее нужное целое число. Например, следующий оператор поместит в метку `lbOutput` дату, соответствующую текущей дате плюс 3 недели:

```
lbOutput.Caption := DateToStr(Date + 21);
```

Чуть сложнее с исчислением времени. Например, чтобы добавить к текущему времени полтора часа, можно использовать любое из выражений:

```
Time + StrToTime('1:30')
Time + 1.5/24
```

Замечу, что специализированные функции `IncXXXX` (см. приложение Б) меняют указанную дату-время на нужное количество лет, месяцев, недель, дней, часов, минут, секунд, миллисекунд:

```
lbOutput.Caption := DateToStr(IncDay(Date, 21));
IncHour(IncMinute(Time, 30), 1).
```

С помощью показанного ниже обработчика `bbRunClick` учебной программы вы сможете ввести в текстовое поле любое вещественное число и посмотреть его интерпретацию как значение типа `TDateTime`:

```
procedure TfmExample.bbRunClick(Sender: TObject);
{Показывает вещественное число, введенное в поле edInput,
в формате даты-времени}
var
    k: Double;
begin
    // Преобразуем строковый ввод в вещественное число
    try
        k := StrToFloat(edInput.Text);
    except
        ShowMessage('Ошибка записи вещественного числа');
        edInput.SelectAll;
        Exit
    end;
    // Показываем результат как дату-время
    lbOutput.Caption := FormatDateTime
        ('dd.mm.yyyy hh:mm:ss', k);
    // Возвращаем фокус ввода в edInput
    edInput.SetFocus;
end;
```

Структурированные типы

Любой из структурированных типов (а в Delphi их четыре: массивы, записи, множества и файлы) характеризуется множественностью образующих этот тип эле-

ментов. Каждый элемент, в свою очередь, может принадлежать структурированному типу, что позволяет говорить о возможной вложенности типов. В Delphi допускается произвольная глубина вложенности типов, однако суммарная длина любого из них во внутреннем представлении не должна превышать 2 Гбайт.

В целях совместимости со стандартным языком Паскаль в Delphi разрешается перед описанием структурированного типа ставить зарезервированное слово **packed**, предписывающее компилятору по возможности экономить память, отводящую под объекты структурированного типа.

Массивы

Массивы в Delphi во многом схожи с аналогичными типами данных в других языках программирования. Отличительная особенность массивов заключается в том, что все их компоненты — данные одного типа (возможно, структурированного). Эти компоненты можно легко упорядочить и обеспечить доступ к любому из них простым указанием его порядкового номера, например:

```
type
  digit = array [0..9] of Char;
  matrix = array [byte] of Single;
var
  m: matrix;
  d: digit;
  i: integer;
begin
  ...
  m[17] := ord(d[i-1])/10;
  ...
end.
```

Описание типа массива задается следующим образом:

```
<имя типа> = array [ <сп.инд.типов> ] of <тип>;
```

Здесь <имя типа> — правильный идентификатор; **array**, **of** — зарезервированные слова (массив, из); <сп.инд.типов> — список из одного или нескольких индексных типов, разделенных запятыми; квадратные скобки, обрамляющие список, — требование синтаксиса; <тип> — любой тип Delphi.

В качестве индексных типов в Delphi можно использовать любые порядковые типы, имеющие мощность не более 2 Гбайт (то есть кроме LongWord и Int64)

Определить переменную как массив можно и непосредственно при описании этой переменной без предварительного описания типа массива, например:

```
var
  a, b: array [1..10] of Real;
```

Обычно в качестве индексного типа используется тип-диапазон, в котором задаются границы изменения индексов. Так как тип <тип>, идущий в описании массива за словом **of**, — любой тип Delphi, то он может быть, в частности, и другим массивом, например:

```
type
```

```
mat = array [0..5] of array [-2..2] of array [Char] of
  Byte;
```

Такую запись можно заменить более компактной:

type

```
mat = array [0..5, -2..2, char] of Byte;
```

Глубина вложенности структурированных типов вообще, а, следовательно, и массивов произвольная, поэтому количество элементов в списке индексных типов (размерность массива) не ограничено, однако суммарная длина внутреннего представления любого массива не может быть больше 2 Гбайт. В памяти ПК элементы массива следуют друг за другом так, что при переходе от младших адресов к старшим наиболее быстро меняется самый правый индекс массива. Например:

```
var
  a: array [1..2, 1..2] of Byte;
begin
  a[1,1] := 1;
  a[2,1] := 2;
  a[1,2] := 3;
  a[2,2] := 4;
end.
```

Для такого массива в памяти последовательно друг за другом будут расположены байты со значениями 1, 3, 2, 4. Это обстоятельство может оказаться важным при использовании стандартной процедуры копирования памяти `MoveMemory`.

В Delphi можно одним оператором присваивания передать все элементы одного массива другому массиву того же типа, например:

```
var
  a, b: array [1..5] of Single;
begin
  ...
  a := b;
  ...
end.
```

После этого присваивания все пять элементов массива `A` получат те же значения, что и элементы в массиве `B`. Замечу, что следующее объявление создаст *разные* типы массивов:

```
var
  a: array [1..5] of Single;
  b: array [1..5] of Single;
```

Поэтому показанный ниже оператор вызовет сообщение об ошибке:

```
a := b;
```

Над массивами не определены операции отношения. Нельзя, например, записать:

```
if a = b then ...
```

Сравнить два массива можно поэлементно, например:

```

var
  a, b: array [1..5] of Single;
  eq  : Boolean;
  i   : Byte;
begin
  ...
  eq := True;
  for i := 1 to 5 do
    if a[i] <> b[i] then
      eq := False;
  if eq then
    ...
end.

```

Динамические массивы

В версии Delphi 4 впервые введены так называемые динамические массивы. При объявлении динамических массивов в программе не следует указывать границы индексов:

```

var
  A: array of Integer;
  B: array of array of Char;
  C: array of array of array of Real;

```

В этом примере динамический массив A имеет одно измерение, массив B — два и массив C — три измерения. Распределение памяти и указание границ индексов по каждому измерению динамических массивов осуществляется в ходе выполнения программы путем *инициализации* массива с помощью функции `SetLength`. В ходе выполнения следующего оператора одномерный динамический массив A будет *инициализирован*, то есть получит память, достаточную для размещения трех целочисленных значений:

```
SetLength(A, 3);
```

Нижняя граница индексов по любому измерению динамического массива всегда равна 0, поэтому верхней границей индексов для A станет значение 2.

Фактически идентификатор динамического массива ссылается на указатель, содержащий адрес первого байта памяти, выделенной для размещения массива. Поэтому для освобождения этой памяти достаточно присвоить идентификатору значение **NIL** (другим способом является использование процедуры `Finalize`):

```

var
  A, B: array of Integer;
begin
  // Распределяем память:
  SetLength(A, 10);
  SetLength(B, 20);
  // Используем массивы:
  ...

```



```

// Освобождаем память :
A := NIL;
Finalize (B);
end;

```

При изменении длины уже инициализированного динамического массива по какому-либо его измерению сначала резервируется нужная для размещения нового массива память, затем элементы старого массива переносятся в новый, после чего освобождается память, выделенная прежнему массиву. Чтобы сократить дополнительные затраты времени, связанные с изменением границ большого динамического массива, следует сразу создать массив максимальной длины.

В многомерных массивах сначала устанавливается длина его первого измерения, затем второго, третьего и т. д. Например, вот как создается динамический массив 3×3 :

```

var
  A: array of array of Integer;
// Двумерный динамический массив
begin
// Устанавливаем длину первого измерения
// (количество столбцов):
  SetLength (A, 3);
// Задаем длину каждого столбца:
  SetLength (A[0], 3);
  SetLength (A[1], 3);
  SetLength (A[2], 3);
  ...
end;

```

Замечу, что `SetLength` — одна из немногих стандартных подпрограмм, которая допускает обращение с переменным количеством параметров. Вместо четырех строк в предыдущем примере можно использовать одну:

```
SetLength (A, 3, 3);
```

Обратите внимание: в отличие от обычных массивов стандартного языка Паскаль (и Delphi), динамические массивы могут иметь *разную* длину по второму и следующим измерениям. В предыдущем примере определен квадратный массив 3×3 . Однако ничто не мешает нам создать, например, треугольный массив (обращаться к подпрограмме `SetLength` с количеством параметров более трех нельзя):

```

SetLength (A, 3);
// Задаем длину каждого столбца:
SetLength (A[0], 3);
SetLength (A[1], 4);
SetLength (A[2], 5);

```

В многомерных динамических массивах каждый элемент любого из $N-1$ измерений (N — количество измерений) представляет собой динамический массив и, следовательно, нуждается в инициализации. Вот как, например, можно инициализировать вещественный кубический массив $3 \times 3 \times 3$:

```

var
  A: array of array of array of Real;
  i, j: Integer;
begin
  SetLength(A, 3);
  for i := 0 to 2 do
    begin
      SetLength(A[i], 3);
      for j := 0 to 2 do
        SetLength(A[i, j], 3);
      end;
    end;
  end;
end;

```

Записи

Запись — это структура данных, состоящая из фиксированного количества компонентов, называемых полями записи. В отличие от массива, компоненты (поля) записи могут быть *разного* типа. Чтобы можно было ссылаться на тот или иной компонент записи, поля именуются.

Структура объявления типа записи такова:

```
<имя типа> = record <сп.полей> end;
```

Здесь <имя типа> — правильный идентификатор; **record, end** — зарезервированные слова (запись, конец); <сп.полей> — список полей; представляет собой последовательность *разделов записи*, между которыми ставится точка с запятой.

Каждый раздел записи состоит из одного или нескольких идентификаторов полей, отделенных друг от друга запятыми. За идентификатором (идентификаторами) ставится двоеточие и описание типа поля (полей), например:

```

type
  Birthday = record
    Day, Month: Byte;
    Year      : Word
  end;
var
  a, b : Birthday;
...

```

В этом примере тип Birthday (день рождения) есть запись с полями Day, Month и Year (день, месяц и год); переменные A и B содержат записи типа Birthday.

Как и в массиве, значения переменных типа записи можно присваивать другим переменным того же типа, например:

```
a := b;
```

К каждому из компонентов записи можно получить доступ, если использовать составное имя, то есть указать имя переменной, затем точку и имя поля:

```
a.day := 27;
```

```
b.year := 2002;
```

Для вложенных полей приходится продолжать уточнения:

```
type
  BirthDay = record
    Day, Month: Byte;
    Year      : Word
  end;
var
  c : record
    Name: String;
    Bd  : BirthDay
  end;
begin
  ...
  if c.Bd.Year = 1989 then ...
end.
```

Чтобы упростить доступ к полям записи, используется оператор присоединения **with**:

```
with <переменная> do <оператор>;
```

Здесь **with**, **do** — зарезервированные слова (с, делать); <переменная> — имя переменной типа записи, за которым, возможно, следует список вложенных полей; <оператор> — любой оператор Delphi.

Например:

```
c.Bd.Month := 9;
```

Это эквивалентно любому из следующих трех операторов:

```
with c.Bd do Month := 9;
with c do with Bd do Month := 9;
with c, Bd do Month := 9;
```

Delphi разрешает использовать записи с так называемыми вариантными полями, например:

```
type
  Forma = record
    Name: String;
    case byte of
      0: (BirthPlace: String [40]);
      1: (Country   : String [20];
         EntryPort : String [20];
         EntryDate  : 1..31;
         ExitDate   : 1..31)
  end;
```

В этом примере тип *Forma* определяет запись с одним фиксированным полем *Name* и вариантной частью, которая задается предложением **case ... of**. Вариантная часть состоит из нескольких вариантов (в примере — из двух вариантов: 0 и 1).

Каждый вариант определяется константой выбора, за которой следует двоеточие и список полей, заключенный в круглые скобки. В любой записи может быть только одна вариантная часть, и, если она есть, она должна располагаться за всеми фиксированными полями.

Замечательной особенностью вариантной части является то обстоятельство, что все заданные в ней варианты «накладываются» друг на друга, то есть каждому из них выделяется одна и та же область памяти. Это открывает дополнительные возможности преобразования типов, например:

```
var
  Mem4: record
    case Byte of
      0: (by: array [0..3] of Byte);
      1: (wo: array [0..1] of Word);
      2: (lo: LongInt);
    end;
```

В этом примере запись Mem4 имеет три варианта, каждый из которых занимает в памяти один и тот же участок из 4 байт. В зависимости от того, к какому полю записи мы обращаемся в программе, этот участок может рассматриваться как массив из 4 байт (поле by), массив из двух целых типа Word (поле wo) или, наконец, как одно целое число типа LongInt (поле lo). Например, этой записи можно сначала присвоить значение как длинному целому, а затем проанализировать результат по байтам или словам:

```
var
  x : Word;
  xb: Byte;
  xl: LongInt;
begin
  ...
  with m do
    begin
      lo := Trunc(2*pi*x);
      if wo[1] = 0 then
        if by[1] = 0 then
          xb := x[0]
        else
          x := wo[0]
        else
          xl := lo
      end;
    ...
  end.
```

В вариантной части нельзя указывать поля с управляемым временем жизни (подробнее об этом см. раздел «Типы с управляемым временем жизни» далее в этой главе).

Предложение **case ... of**, открывающее вариантную часть, внешне похоже на соответствующий оператор выбора, но на самом деле лишь играет роль своеобразного служебного слова, обозначающего начало вариантной части. Именно поэтому в конце вариантной части не следует ставить слово **end** как пару к **case ... of** (поскольку вариантная часть — всегда последняя в записи, за ней все же стоит **end**, но лишь как пара к **record**). Ключ выбора в предложении **case ... of** фактически игнорируется компилятором: единственное требование, предъявляемое к нему в Delphi, состоит в том, чтобы ключ определял некоторый стандартный или предвзятительно объявленный порядковый тип.

Имена полей должны быть уникальными в пределах той записи, где они объявлены, однако если записи содержат поля-записи, то есть вложены одна в другую, имена могут повторяться на разных уровнях вложения.

Множества

Множества — это наборы логически связанных друг с другом объектов. Характер связей между объектами лишь подразумевается программистом и никак не контролируется Delphi. Количество элементов, входящих во множество, может меняться в пределах от 0 до 256 (множество, не содержащее элементов, называется *пустым*). Именно непостоянством количества своих элементов множества отличаются от массивов и записей.

Два множества считаются эквивалентными тогда и только тогда, когда все их элементы одинаковы, причем порядок следования элементов во множестве безразличен. Если все элементы одного множества входят также и в другое, говорят о включении первого множества во второе. Пустое множество включается в любое другое.

Пример определения и задания множеств:

```
type
  digitChar = set of '0'..'9';
  digit      = set of 0..9;
var
  s1,s2,s3 : digitChar;
  s4,s5,s6 : digit;
begin
  ....
  s1 := ['1', '2', '3'];
  s2 := ['3', '2', '1'];
  s3 := ['2', '3'];
  s4 := [0..3, 6];
  s5 := [4, 5];
  s6 := [3..9];
  ...
end.
```

В этом примере множества S1 и S2 эквивалентны, а множество S3 включено в S2, но не эквивалентно ему.

Описание типа множества имеет вид:

```
<имя типа> = set of <базовый тип>;
```

Здесь <имя типа> — правильный идентификатор; **set, of** — зарезервированные слова (множество, из); <базовый тип> — базовый тип элементов множества, в качестве которого может использоваться любой порядковый тип, кроме Word, Integer, LongInt, Int64.

Для задания множества используется так называемый *конструктор множества*: список спецификаций элементов множества, отделенных друг от друга запятыми; список обрамляется квадратными скобками. Спецификациями элементов могут быть константы или выражения базового типа, а также тип-диапазон того же базового типа.

Над множествами определены следующие операции:

- ✱ пересечение множеств (*); результат содержит элементы, общие для обоих множеств; например, $S4 * S6$ содержит [3], $S4 * S5$ — пустое множество (см. выше);
- ✱ объединение множеств (+); результат содержит элементы первого множества, дополненные недостающими элементами из второго множества:
 - $S4 + S5$ содержит [0, 1, 2, 3, 4, 5, 6];
 - $S5 + S6$ содержит [3, 4, 5, 6, 7, 8, 9];
- ✱ разность множеств (-); результат содержит элементы из первого множества, которые не принадлежат второму:
 - $S6 - S5$ содержит [3, 6, 7, 8, 9];
 - $S4 - S5$ содержит [0, 1, 2, 3, 6];
- ✱ проверка эквивалентности (=); возвращает True, если оба множества эквивалентны;
- ✱ проверка неэквивалентности (<>); возвращает True, если оба множества неэквивалентны;
- ✱ проверка вхождения (<=); возвращает True, если первое множество включено во второе;
- ✱ проверка вхождения (>=); возвращает True, если второе множество включено в первое;
- ✱ проверка принадлежности (**in**); в этой бинарной операции первый элемент — выражение, а второй — множество одного и того же типа; возвращает True, если выражение имеет значение, принадлежащее множеству:
 - $3 \text{ in } s6$ возвращает True;
 - $2 * 2 \text{ in } s1$ возвращает False.

Дополнительно к этим операциям можно использовать две процедуры.

Процедура Include включает новый элемент во множество. Обращение к процедуре:

```
Include (S, I)
```

Здесь S — множество, состоящее из элементов базового типа TSetBase; I — элемент типа TSetBase, который необходимо включить во множество.

Процедура Exclude исключает элемент из множества. Обращение:

```
Exclude (S, I)
```

Параметры обращения — такие же, как у процедуры Include.

В отличие от операций + и –, реализующих аналогичные действия над двумя множествами, процедуры оптимизированы для работы с одиночными элементами множества и поэтому отличаются высокой скоростью выполнения.

В следующем примере, иллюстрирующем приемы работы с множествами, реализуется алгоритм выделения из первой сотни натуральных чисел всех простых чисел¹. В его основе лежит прием, известный под названием «решето Эратосфена». В соответствии с этим алгоритмом вначале формируется множество BeginSet, состоящее из всех целых чисел в диапазоне от 2 до N. Во множество PrimerSet (оно будет содержать искомые простые числа) помещается 1. Затем циклически повторяются следующие действия.

1. Взять из BeginSet первое входящее в него число Next и поместить его в PrimerSet.
2. Удалить из BeginSet число Next и все другие числа, кратные ему, то есть $2 * \text{Next}$, $3 * \text{Next}$ и т. д.

Цикл повторяется до тех пор, пока множество BeginSet не станет пустым.

Эту программу нельзя использовать для произвольного значения N, так как в любом множестве не может быть больше 256 элементов.

```
procedure TfmExample.bbRunClick(Sender: TObject);
// Выделение всех простых чисел из первых N целых
const
    N = 255; // Количество элементов исходного множества
type
    SetOfNumber = set of 1..N;
var
    n1, Next, i: Word;           // Вспомогательные переменные
    BeginSet,                 // Исходное множество
    PrimerSet: SetOfNumber; // Множество простых чисел
    S: String;
begin
    BeginSet := [2..N];           // Создаем исходное множество
    PrimerSet := [1];             // Первое простое число
    Next := 2;                   // Следующее простое число
    while BeginSet <> [] do // Начало основного цикла
    begin
        n1 := Next; // n1 - число, кратное очередному простому
                     // (Next)
        // Цикл удаления из исходного множества непростых чисел:
        while n1 <= N do
        begin
            Exclude(BeginSet, n1);
```

¹ Простыми называются целые числа, которые не делятся без остатка на любые другие целые числа, кроме единицы и себя. К простым относятся числа 1, 2, 3, 5, 7, 11, 13 и т. д.

```

    n1 := n1 + Next      // Следующее кратное
end;                    // Конец цикла удаления
Include(PrimerSet, next);
// Получаем следующее простое, которое есть первое
// число, не вычеркнутое из исходного множества
repeat
    inc(Next)
until (Next in BeginSet) or (Next > N)
end; // Конец основного цикла
// Выводим результат:
S := '1';
for i := 2 to N do
    if i in PrimerSet then
        S := S + ', ' + IntToStr(i);
mmOutput.Lines.Add(S)
end;

```

Перед тем как закончить рассмотрение множеств, полезно провести небольшой эксперимент. Измените описание типа `SetOfNumber` следующим образом:

```

type
    SetOfNumber = set of 1..1;

```

Еще раз запустите программу из предыдущего примера. На экран будет выведено:

```
1, 3, 5, 7
```

Множества `BeginSet` и `PrimerSet` состоят теперь из одного элемента, а программа сумела поместить в них не менее семи!

Секрет прост: внутреннее устройство множества таково, что каждому его элементу ставится в соответствие один двоичный разряд (один бит); если элемент включен во множество, соответствующий разряд имеет значение 1, в противном случае — 0. В то же время минимальной единицей памяти является один байт, содержащий 8 бит, поэтому компилятор выделил множествам по одному байту, и в результате мощность каждого из них стала равна 8 элементам. Максимальная мощность множества — 256 элементов. Для таких множеств компилятор выделяет по 16 смежных байтов.

И еще один эксперимент: измените диапазон базового типа на `1..256`. Хотя мощность этого типа составляет 256 элементов, при попытке компиляции программы компилятор сообщит об ошибке (множества могут иметь не более 256 элементов):

```
Sets may have at most 256 elements
```

Причина — нумерация элементов множества начинается с нуля, независимо от объявленной в программе нижней границы. Компилятор разрешает использовать в качестве базового типа целочисленный тип-диапазон с минимальной границей 0 и максимальной 255 или любой перечисленный тип не более чем с 256 элементами (максимальная мощность перечисленного типа — 65 536 элементов).

Строки

Для обработки текстов в Delphi используются следующие типы:

- короткая строка ShortString или **String**[N], где N ≤ 255;
- длинная строка **String** или AnsiString;
- широкая строка WideString;
- заканчивающаяся нулем строка PChar.

Общим для этих типов является то, что каждая строка трактуется как одномерный массив символов, количество элементов в котором может меняться в работающей программе: для **String**[N] длина строки меняется от 0 до N, для **String**, WideString и PChar — от 0 до 2 Гбайт.

В стандартном языке Паскаль используются только короткие строки **String**[N]. В памяти такой строке выделяется N+1 байт, первый байт содержит текущую длину строки, а сами символы располагаются, начиная со 2-го по счету байта. Поскольку для длины строки в этом случае отводится один байт, максимальная длина короткой строки не может превышать 255 символов. Для объявления короткой строки максимальной длины предназначен стандартный тип ShortString (эквивалент **String**[255]).

В Windows широко используются заканчивающиеся нулем строки, представляющие собой цепочки символов, ограниченные символом #0. Максимальная длина такой строки лимитируется только доступной памятью и может быть очень большой.

В 32-разрядных версиях Delphi введен новый тип **String**, сочетающий в себе удобства обоих типов. При работе с этим типом память выделяется по мере необходимости (динамически) и ограничена имеющейся в распоряжении программы доступной памятью.

Для совместимости с компонентами, основывающимися на OLE-технологии, в 32-разрядных версиях Delphi введены также широкие строки, объявленные стандартным типом WideString. По своим свойствам они идентичны длинным строкам **String**, но отличаются от них тем, что для представления каждого символа используются не один, а два байта.

Примеры объявлений строковых типов:

var

```
ssS: String[250]; // Короткая строка длиной до 250 символов
ssMax: ShortString; // Короткая строка длиной до
                    // 255 символов
stS: String;      // Длинная строка
swS: WideString; // Широкая строка
pcS: PChar;       // Ссылка на заканчивающуюся нулем строку
acS: array [0..1000] of Char; // Заканчивающаяся нулем
                               // строка длиной до
                               // 1000 символов
```

При объявлении переменной ssS компилятор выделяет для ее размещения 250 + 1 = 251 байт и помещает в первый байт 0 — текущую длину строки. Рассмотрим следующий фрагмент программы:

```

procedure TfmExample.bbRunClick(Sender: TObject);
var
    ssS: String[250];
begin
    ssS := 'Строка символов';
    ssS[6] := 'и'; // Символы в строке нумеруются, начиная с 1
    lbOutput.Caption := ssS; // Выводится строка "Строки
    символов"
end;

```

При выполнении этого фрагмента сначала в переменную `ssS` будет помещена цепочка символов `Строка символов`, причем 1-й байт получит значение 15 (количество символов в строке). После выполнения второго оператора символ с индексом 6 (индексация байтов начинается с 0, но поскольку первый байт содержит текущую длину, первый *символ* в строке имеет индекс 1) будет заменен символом «и», и в переменной окажется цепочка `Строки символов`.

Совершенно другим будет механизм работы с памятью при объявлении длинной строки `stS`: компилятор выделит для переменной 4 байта, достаточные для размещения *номера* той ячейки памяти, начиная с которой будет фактически располагаться символьная строка. Говорят, что `stS` *ссылается* на строку. Такого рода переменные-ссылки называются указателями и обсуждаются ниже в разделе «Указатели и динамическая память». Рассмотрим следующий обработчик:

```

procedure TfmExample.bbRunClick(Sender: TObject);
var
    stS, stSS: String;
begin
    stS := 'Строка символов';
    stSS := stS;
    stS := 'Это - ' + stS;
    stS[7] := 'с'; // Символы в строке нумеруются, начиная с 1
    lbOutput.Caption := stS;
    // Выводится "Это - строка символов"
end;

```

При выполнении первого оператора такого обработчика программа (а не компилятор!) определит длину цепочки символов `Строка символов`, обратится к ядру операционной системы с требованием выделить для нее участок памяти длиной $15 + 5 = 20$ байт, поместит в переменную `stS` номер первого выделенного байта¹ и, начиная с него, разместит в этом участке цепочку символов, завершив ее завершающим нулем и 4-байтным счетчиком ссылок. Такое размещение на этапе прогона программы называется динамическим, в то время как размещение на этапе компиляции — статическим. Счетчик ссылок играет важную роль в механизме работы с памятью. С его помощью реализуется «кэширование» памяти:

```
stSS := stS;
```

¹ На самом деле в `stS` запоминается *дескриптор* выделенного участка памяти (см. главу 9).

При выполнении этого оператора память для размещения *значения* переменной stSS не выделяется, в переменную stSS помещается содержимое указателя stS, а счетчик ссылок в связанной с ним памяти увеличивается на единицу. Таким образом, оба указателя будут ссылаться на одну и ту же область памяти, счетчик ссылок которой будет содержать значение 2. При выполнении следующего оператора счетчик ссылок уменьшается на единицу, выделяется новая область памяти длиной $20 + 6 = 26$ байт, указатель на эту область помещается в stS, а в саму память переписывается цепочка символов Это – Строка символов, завершающий ноль и содержащий единицу счетчик ссылок:

```
stS := 'Это – ' + stS;
```

Теперь переменные stS и stSS ссылаются на разные участки памяти, счетчики ссылок которых содержат по единице. Выделенная для размещения строки **String** область памяти освобождается, если ее счетчик ссылок становится равным нулю.

Похожим образом осуществляется работа с памятью при объявлении переменной pcS типа PChar: компилятор считает эту переменную указателем и выделяет для нее 4 байта:

```
procedure TfmExample.bbRunClick(Sender: TObject);
var
  pcS: PChar;
begin
  pcS := 'Строка символов';
  pcS[5] := 'и'; {Символы в строке нумеруются, начиная с 0}
  lbOutput.Caption := pcS; // Выводится "Строки символов"
end;
```

Программа потребует от операционной системы $15 + 1 = 16$ байт, разместит в памяти цепочку символов с завершающим ее нулем и поместит адрес выделенного участка памяти в pcS.

ПРИМЕЧАНИЕ

В стандартном языке Паскаль при обращении к области памяти, на которую ссылается указатель, требуется за именем указателя ставить специальный символ «^». В Delphi интенсивно используется динамическая память, поэтому это жесткое требование смягчено: в большинстве случаев (и при обращении к типу PChar в том числе) символ «^» ставить не следует.

И, наконец, последнее объявление acS как массива символов. В Delphi считается совместимым с PChar одномерный массив символов с нулевой нижней границей. В отличие от pcS память для такой переменной выделяется статически (в нашем примере компилятор выделит в сегменте данных для переменной acS 1001 байт). Для размещения в acS завершающейся нулем цепочки символов используется процедура StrCopy:

```
procedure TfmExample.bbRunClick(Sender: TObject);
var
  acS: array [0..1000] of Char;
```

```

begin
  StrCopy(acS, 'Строка символов');
  acS[5] := 'и'; { Символы в строке нумеруются, начиная с 0 }
  lbOutput.Caption := acS; // Выводится "Строки символов"
end;

```

Необходимость в строках, завершающихся нулем, возникает только при прямом обращении к API-функциям Windows. При работе с компонентами Delphi в основном используются более удобные длинные строки, которые рассматриваются ниже.

Еще несколько слов о широких строках. 32-разрядные версии Windows используют три сорта символов: однобайтный символ ANSI, двухбайтный символ и символ Unicode. Однобайтный символ связан с одним из 256 возможных значений, которые трактуются в зависимости от установленной в Windows национальной страницы (для кириллицы используется страница 1251). 256 символов вполне достаточно для отображения национального алфавита любого европейского языка. Для отображения алфавитов некоторых азиатских языков этого недостаточно. В этом случае используется двухбайтный символ, в котором младший байт обычно кодируется семибитным ASCII-кодом, а старший указывает, как должен трактоваться этот код (каким символом он будет изображаться в документе или на экране). Символ Unicode в памяти занимает одно слово, которое имеет 65 536 возможных значений. Специальная международная комиссия по Unicode выработала соглашение, позволяющее с помощью этого кода представить все символы всех языков мира. Двухбайтные символы и символы Unicode объявляются стандартным типом `WideChar`, а составленные из них строки — типом `WideString`. Все Windows-программы, использующие OLE-технологии обмена строками, должны кодировать символы в соответствии с Unicode.

Несмотря на разницу во внутреннем представлении, короткие строки `ShortString` и длинные строки `String` имеют для программиста одинаковые свойства.

Текущую длину строки можно получить с помощью функции `Length`. Например, следующий оператор уничтожает все ведомые (хвостовые) пробелы:

```

while (Length(stS) <> 0) and (stS[Length(stS)] = ' ') do
  SetLength(stS, Length(stS) - 1);

```

В этом примере стандартная процедура `SetLength` устанавливает новую длину строки. К строкам можно применять операцию сцепления (+), например:

```

stS := 'a' + 'b'; // stS содержит "ab"
stS := stS + 'c'; // stS содержит "abc"

```

Если длина строки превысит максимально допустимую длину `N` короткой строки, то «лишние» символы отбрасываются. Следующая программа, например, выведет символ «!»:

```

procedure TfmExamlpe.bbRunClick(Sender: TObject);
var
  ssS: String[1];
begin
  ssS := '123';

```

```
lbOutput := ssS;
end;
```

Операции отношения (=, <>, >, <, >= и <=) выполняются над двумя строками посимвольно, слева направо с учетом внутренней кодировки символов. Если одна строка меньше другой по длине, недостающие символы короткой строки заменяются значением #0.

Следующие операции отношения дадут значение True:

```
'.' < '.'
'A' > '1'
'Object' < ' Delphi'
'Пас' > 'Pascal'
```

Все остальные действия над строками и символами реализуются с помощью стандартных процедур и функций, указанных в табл. 6.8 (см. также приложение Б).

Таблица 6.8. Подпрограммы для работы со строками

Подпрограмма	Описание
function AnsiLowerCase (const S: String): String;	Возвращает исходную строку S, в которой все прописные буквы заменены строчными в соответствии с национальной кодировкой Windows (то есть с учетом кириллицы)
function AnsiUpperCase (const S: String): String;	Возвращает исходную строку S, в которой все строчные буквы заменены прописными в соответствии с национальной кодировкой Windows
function Concat(S1 [, S2, ..., SN]: String): String;	Возвращает строку, представляющую собой сцепление строк-параметров S1, S2, ..., SN
function Copy(St: String; Index, Count: Integer): String;	Из строки St копирует Count символов, начиная с символа с номером Index
procedure Delete(St: String; Index, Count: Integer);	Удаляет Count символов из строки St, начиная с символа с номером Index
procedure Insert(SubSt: String; St, Index: Integer);	Вставляет подстроку SubSt в строку St, начиная с символа с номером Index
function Length(St: String): Integer;	Возвращает текущую длину строки St
function LowerCase(const S: String): String;	Возвращает исходную строку S, в которой все латинские прописные буквы заменены строчными
procedure OleStrToStrVar (Source: PWideChar; var Dest: String);	Копирует широкую (двухбайтную) строку в обычную строку Delphi

Подпрограмма	Описание
function Pos(SubSt, St: String): Integer;	Отыскивает в строке St первое вхождение подстроки SubSt и возвращает номер позиции, с которой она начинается. Если подстрока не найдена, возвращается ноль
procedure SetLength(St: String; NewLength: Integer);	Устанавливает новую (меньшую) длину NewLength строки St. Если NewLength больше текущей длины строки, обращение к SetLength игнорируется
function StringOfChar(Ch: Char; Count: Integer): String;	Создает строку, состоящую из Count раз повторенного символа Ch
function StringToOleStr(const Source: String): PWideChar;	Копирует обычную строку в двухбайтную
function StringToWideChar(const Source: String; Dest: PWideChar; DestSize: Integer): PWideChar;	Преобразует обычную строку в строку с символами Unicode
function UpperCase(const S: String): String;	Возвращает исходную строку S, в которой все строчные латинские буквы заменены прописными
<i>Подпрограммы преобразования строк в другие типы</i>	
function StrToCurr(St: String): Currency;	Преобразует символы строки St в целое число типа Currency. Строка не должна содержать ведущих или ведомых пробелов
function StrToDate(St: String): TDateTime;	Преобразует символы строки St в дату. Строка должна содержать два или три числа, разделенных правильным для Windows разделителем даты (в русифицированной версии таким разделителем является точка). Первое число — правильный день, второе — правильный месяц. Если указано третье число, оно должно задавать год в формате XX или XXXX. Если символы года отсутствуют, дата дополняется текущим годом. Например, оператор DateToStr(StrToDate('28.06')) даст строку '28.06.02' (см. ниже пояснения)
function StrToDateTime(St: String): TDateTime;	Преобразует символы строки St в дату и время. Строка должна содержать правильную дату (см. StrToDate) и правильное время (см. StrToTime), разделенные пробелом, например: StrToDateTime('28.06 18:23')
function StrToFloat(St: String): Extended;	Преобразует символы строки St в вещественное число. Строка не должна содержать ведущих или ведомых пробелов
function StrToInt(St: String): Integer;	Преобразует символы строки St в целое число. Строка не должна содержать ведущих или ведомых пробелов

Таблица 6.8 (продолжение)

Подпрограмма	Описание
function StrToIntDef (St: String ; Default: Integer): Integer;	Преобразует символы строки St в целое число. Если строка не содержит правильного представления целого числа, возвращается значение Default
function StrToIntRange (St: String ; Min, Max: LongInt): LongInt;	Преобразует символы строки St в целое число и возбуждает исключение ERangeError, если число выходит из заданного диапазона Min...Max
function StrToTime (St: String): TDateTime;	Преобразует символы строки St во время. Строка должна содержать два или три числа, разделенных правильным для Windows разделителем времени (для русифицированной версии таким разделителем является двосточие). Числа задают часы, минуты и, возможно, секунды. За последним числом через пробел могут следовать символы «am» или «pm», указывающие на 12-часовой формат времени
procedure Val (St: String ; var X; Code: Integer);	Преобразует строку символов St во внутреннее представление целой или вещественной переменной X, которое определяется типом этой переменной. Параметр Code содержит ноль, если преобразование прошло успешно, и тогда в X помещается результат преобразования, в противном случае он содержит номер позиции в строке St, где обнаружен ошибочный символ, и в этом случае содержимое X не меняется. В строке St могут быть ведущие и/или ведомые пробелы. Если St содержит символьное представление вещественного числа, разделителем целой и дробной частей должна быть точка независимо от того, каким символом этот разделитель указан в Windows
<i>Подпрограммы обратного преобразования</i>	
function DateTimeToStr (Value: TDateTime): String ;	Преобразует дату и время из параметра Value в строку символов
procedure DateTimeToString St: String ; Format: String ; Value: TDateTime);	Преобразует дату и время из параметра Value (var в строку St в соответствии со спецификаторами параметра Format (см. пояснения ниже)
function DateToStr (Value: TDateTime): String ;	Преобразует дату из параметра Value в строку символов
function FloatToStr (Value: Extended): String ;	Преобразует вещественное значение Value в строку символов
function FloatToStrF (Value: Extended; Format: TFloatFormat; Precision, Digits: Integer): String ;	Преобразует вещественное значение Value в строку символов с учетом параметра Format и параметров Precision и Digits (см. пояснения ниже)

Подпрограмма	Описание
function Format(const Format: String ; const Args: array of const): String ;	Преобразует произвольное количество аргументов открытого массива Args в строку в соответствии со спецификаторами параметра Format (см. пояснения ниже)
function FormatDateTime (Format: String ; Value: TDateTime): String ;	Преобразует дату и время из параметра Value в строку символов в соответствии со спецификаторами параметра Format (см. пояснения ниже)
function FormatFloat (Format: String ; Value: Extended): String ;	Преобразует вещественное число Value в строку символов в соответствии со спецификаторами параметра Format (см. пояснения ниже)
function IntToHex(Value: Integer; Digits: Integer): String ;	Преобразует целое число Value в строку символьного представления шестнадцатеричного формата: Digits — минимальное количество символов в строке
function IntToStr(Value: Integer): String ;	Преобразует целое число Value в строку символов
procedure Str(X [:Width [:Decimals]]; var St: String);	Преобразует число X любого вещественного или целого типов в строку символов St; параметры Width и Decimals, если они присутствуют, задают формат преобразования: Width определяет общую ширину поля, выделенного под соответствующее символьное представление числа X, а Decimals — количество символов в дробной части (этот параметр имеет смысл только в том случае, когда X — вещественное число)
function TimeToStr(Value: TDateTime): String ;	Преобразует время из параметра Value в строку символов

При преобразовании с помощью функций StrToXXX строка может содержать недопустимые для типа XXX символы. В этом случае возбуждается исключительная ситуация EConvertError. Однако при использовании StrToIntDef исключение не возникает, и функция возвращает умалчиваемое значение. При обратных преобразованиях XXXToStr формат получаемой строки зависит от установленных в Windows системных параметров: разделителей даты, времени, целой и дробной частей вещественного числа.

Используемая в процедуре DateTimeToString и в функции FormatDateTime строка Format может содержать символы-спецификаторы (см. приложение Б). Замечу, что любые другие символы, указанные в строке Format, а также заключенные в апострофы или кавычки специальные символы-спецификаторы помещаются в выходную строку без преобразования, поэтому спецификаторы 'h час n мин' дадут строку 19 час 45 мин, а 'hчас "n" мин' — 19 час n мин.

При форматном преобразовании времени-даты или других типов в строку и обратно могут пригодиться системные переменные, приведенные в приложении Б.

Для форматного преобразования вещественных чисел предназначены функции `FloatToStrF` и `FormatFloat`. Первая использует значение `Format` перечисленного типа `TFloatFormat` и два дополнительных параметра — `Precision` и `Digits`. Правила использования параметров функции `FloatToStrF`, а также спецификаторы параметра `Format` в функции `FormatFloat` приведены в приложении Б.

Как и в случае даты/времени, любые другие символы строки `Format`, а также заключенные в апострофы или кавычки специальные символы-спецификаторы помещаются в выходную строку без преобразования: для параметра `Value`, равно-го $\pi \times 1000$ спецификаторы `'#, рубль'` дадут 3 142 рубля.

Мощная функция преобразования `Format` перешла в Delphi из языка Си (в нем она называется `printf`). Она позволяет преобразовать сразу несколько элементов открытого массива аргументов в соответствии с указаниями форматирующей строки. Например, рассмотрим выражение:

```
Format('Строка "%s" содержит %d символов', ['Паскаль', 7])
```

Это выражение даст такой результат:

```
Строка "Паскаль" содержит 7 символов
```

Элементами массива аргументов могут быть константы и/или переменные целого и вещественного типа, строки и указатели.

Форматирующая строка — это произвольная строка, в которую в любом месте можно вставить *форматирующий спецификатор*. Количество форматирующих спецификаторов должно быть не больше количества элементов массива аргументов — в противном случае возникнет исключительная ситуация. Каждому элементу массива аргументов по порядку их перечисления в конструкторе массива функция ставит в соответствие форматирующий спецификатор по порядку его следования в форматирующей строке: первому аргументу — первый спецификатор, второму — второй и т. д. Если количество спецификаторов меньше количества аргументов, «лишние» аргументы игнорируются.

Форматирующий спецификатор всегда начинается символом процента и в общем случае имеет такую структуру (в квадратных скобках указываются необязательные элементы):

```
"%" [index ":" ] ["-"] [width] [". " prec] type
```

Здесь: `index ":"` — индекс открытого массива, с помощью этого элемента можно явно указать аргумент, который будет обрабатывать спецификатор; `"-"` указывает на необходимость прижать отформатированный спецификатором текст к левой границе отведенного для него пространства; `width` — число, определяющее количество символов для обработанного спецификатором текста; если это число меньше требуемого, этот элемент спецификатора игнорируется, если больше — дополняется справа (если есть элемент `"-"`) или слева (если он отсутствует) нужным количеством пробелов; `". " prec` — точность представления целых и вещественных типов; `type` — символ, определяющий тип форматирования (табл. 6.9).

Таблица 6.9. Форматирующие спецификаторы для функции Format

Спецификатор	Описание
d	Целое десятичное число. Если задан параметр <code>prec</code> , то символическое представление должно содержать по меньшей мере <code>prec</code> десятичных цифр: если символическое представление содержит меньше цифр, оно дополняется слева символами 0, если больше — параметр <code>prec</code> игнорируется. Если аргумент не является целым числом, возникает исключительная ситуация
u	Беззнаковое целое число. Используется подобно типу <code>d</code> , но аргумент должен быть положительным числом. Если аргумент — отрицательное целое, результат форматирования непредсказуем
e	Вещественное число в экспоненциальном представлении: символическое представление имеет вид <code>-d, ddddddddddE+ddd</code> . Если число положительное, ведущий минус опускается, а если его модуль меньше 1, знак плюс меняется на минус. Всегда содержит одну цифру целой части, по меньшей мере, одну цифру дробной части и не меньше трех цифр десятичного порядка. Если задан параметр точности, он определяет общее количество цифр до символа <code>E</code> , но не меньше двух: если параметр точности содержит 1 или 0, он заменяется на 2. Если символическое представление содержит больше символов, чем <code>prec</code> , оно округляется по первой отбрасываемой цифре. Умалчиваемое значение параметра точности — 15
f	Вещественное число в виде <code>-ddd, ddd</code> (фиксированное представление). Параметр точности (по умолчанию 2) определяет количество цифр в дробной части. Если он нулевой, выводится только целая часть числа
g	Вещественное число в максимально коротком представлении (экспоненциальном или фиксированном). Параметр точности (по умолчанию 15) определяет максимальное количество значащих разрядов. Если число имеет меньшее количество цифр, оно не дополняется до <code>prec</code> , а если число не имеет дробной части, оно выводится как целое (без запятой)
n	Соответствует фиксированному представлению, но использует символ-разделитель тысяч
m	Денежное представление вещественного числа. Подобно типу <code>n</code> , но справа ставится знак денежной единицы
p	Указатель. Выводит содержимое указателя в виде 8 шестнадцатеричных цифр
s	Аргумент должен быть символом, строкой или строкой с завершающим нулем. Параметр точности, если указан, определяет максимальную длину строки: если строка больше, она усекается, если меньше — параметр точности игнорируется
x	Шестнадцатеричное представление целого числа. Параметр точности определяет минимальное количество шестнадцатеричных цифр (если число меньше, оно дополняется ведущими нулями)

Функция не чувствительна к регистру букв, определяющих тип преобразования. Параметры `index`, `prec` и `width` задаются явно (числами в формирующей строке) или неявно — с помощью символа «*». В этом случае в качестве параметра берется значение очередного аргумента в списке аргументов (он должен быть целым числом). Например, два следующих выражения дадут одинаковый результат:

```
Format('%*.f', [8, 2, 123.456]);
Format('%8.2f', [123.456]);
```

Указатели и динамическая память

Динамическая память

Динамическая память — это оперативная память компьютера, предоставляемая ядром операционной системы программе по ее требованию. Динамическое размещение данных означает использование динамической памяти непосредственно при работе программы. В отличие от этого статическое размещение осуществляется компилятором Delphi в процессе компиляции программы. При динамическом размещении заранее не известны ни тип, ни объем размещаемых данных.

Указатели

Оперативная память ПК представляет собой совокупность ячеек для хранения информации — байтов, каждый из которых имеет собственный номер. Эти номера называются *адресами*, они позволяют обращаться к любому байту памяти. Delphi предоставляет в распоряжение программиста гибкое средство управления динамической памятью — так называемые указатели. *Указатель* — это переменная, которая в качестве своего значения содержит адрес байта памяти. С помощью указателей можно размещать в динамической памяти любой из известных в Delphi типов данных. Лишь некоторые из них (`Byte`, `Char`, `ShortInt`, `Boolean`) занимают во внутреннем представлении один байт, остальные — несколько смежных. Поэтому на самом деле указатель адресует лишь первый байт данных.

Как правило, указатель связывается с некоторым типом данных. Такие указатели будем называть *типизированными*. Для объявления типизированного указателя используется значок `^`, который помещается перед соответствующим типом, например:

```
var
  p1 : ^Integer;
  p2 : ^Real;
type
  PerconPointer = ^PerconRecord;
  PerconRecord = record
    Name : String;
    Job : String;
    Next : PerconPointer
  end;
```

Обратите внимание: при объявлении типа `PerconPointer` мы сослались на тип `PerconRecord`, который предварительно в программе объявлен не был. Как уже отмечалось, в Delphi последовательно проводится в жизнь принцип, в соответствии с которым перед использованием какого-либо идентификатора он должен быть описан. Исключение сделано только для указателей, которые могут ссылаться на еще не объявленный тип данных.

В Delphi можно объявлять указатель и не связывать его при этом с каким-либо конкретным типом данных. Для этого служит стандартный тип `Pointer`, например:

```
var  
  p: Pointer;
```

Указатели такого рода будем называть *нетипизированными*. Поскольку нетипизированные указатели не связаны с конкретным типом, с их помощью удобно динамически размещать данные, структура и тип которых меняются в ходе работы программы.

Как уже отмечалось, значениями указателей являются адреса переменных в памяти, поэтому следовало бы ожидать, что значение одного указателя можно передавать другому. На самом деле это не совсем так. В Delphi можно передавать значения только между указателями, связанными с одним и тем же типом данных. Например, пусть имеет место следующее объявление:

```
var  
  pI1, pI2: ^Integer;  
  pR: ^Real;  
  p: Pointer;
```

В этом случае показанное ниже присваивание вполне допустимо:

```
pI1 := pI2;
```

В то же время следующее присваивание запрещено, поскольку `pI1` и `pR` указывают на *разные* типы данных:

```
pI1 := pR;
```

Это ограничение, однако, не распространяется на нетипизированные указатели, поэтому мы могли бы записать:

```
p := pR;  
pI1 := p;
```

и тем самым достичь нужного результата.

Выделение и освобождение динамической памяти

Вся динамическая память в Delphi рассматривается как сплошной массив байтов, который называется *кучей*.

Память под любую динамически размещаемую переменную выделяется процедурой `New`. Параметром обращения к этой процедуре является типизированный указатель. В результате обращения указатель приобретает значение, соответствующее адресу, начиная с которого можно разместить данные, например:

```

var
  pI, pJ: ^Integer;
  pR: ^Real;
begin
  New (pI);
  New (pR);
  ...
end;

```

После того как указатель приобрел некоторое значение, то есть стал указывать на конкретный физический байт памяти, по этому адресу можно разместить любое значение соответствующего типа. Для этого в операторе присваивания сразу за указателем без каких-либо пробелов ставится значок $\wedge$, например:

```

pJ $\wedge$  := 2;      // В область памяти pJ помещено значение 2
pR $\wedge$  := 2*pi;  // В область памяти pR помещено значение 6.28

```

Таким образом, *значение*, которое адресует указатель, то есть собственно данные, размещенные в куче, обозначаются значком $\wedge$ сразу за указателем. Если за указателем нет значка $\wedge$, то имеется в виду *адрес*, по которому размещены данные. Имеет смысл еще раз задуматься над только что сказанным: значением любого указателя является адрес, а чтобы показать, что речь идет не об адресе, а о тех данных, которые размещены по этому адресу, за указателем ставится значок $\wedge$ (иногда об этом говорят как о *разыменовании* указателя).

Динамически размещенные данные можно использовать в любом месте программы, где это допустимо для констант и переменных соответствующего типа, например:

```

pR $\wedge$  := Sqr (pR $\wedge$ ) + I $\wedge$  - 17;

```

Разумеется, совершенно недопустим следующий оператор, так как *указателю* pR нельзя присвоить значение вещественного выражения:

```

pR := Sqr (pR $\wedge$ ) + I $\wedge$  - 17;

```

Точно так же недопустим оператор:

```

pR $\wedge$  := Sqr (pR);

```

Причина в том, что значением указателя pR является адрес и его (в отличие от того значения, которое размещено по этому адресу) нельзя возводить в квадрат. Ошибочным будет и следующее присваивание, так как вещественным данным, на которые указывает pR, нельзя присвоить значение указателя (адрес):

```

pR $\wedge$  := pJ;

```

Динамическую память можно не только забирать из кучи, но и возвращать обратно. Для этого используется процедура `Dispose`. Например, показанные ниже операторы вернут в кучу память, которая ранее была закреплена за указателями pJ и pR (см. выше).

```

Dispose (pJ);
Dispose (pR);

```

Замечу, что процедура `Dispose (pPtr)` не изменяет значения указателя pPtr, а лишь возвращает в кучу память, ранее связанную с этим указателем. Однако

повторное применение процедуры к свободному указателю приведет к возникновению ошибки периода исполнения. Освободившийся указатель программист может пометить зарезервированным словом **NIL**. Помечен ли какой-либо указатель или нет, можно проверить следующим образом:

```
const
  pR: ^Real = NIL;
begin
  ...
  if pR = NIL then
    New(pR);
  ...
  Dispose(pR);
  pR := NIL;
  ...
end;
```

Никакие другие операции сравнения над указателями не разрешены.

Приведенный выше фрагмент иллюстрирует предпочтительный способ объявления указателя в виде типизированной константы с одновременным присваиванием ему значения **NIL**. Следует учесть, что начальное значение указателя (при его объявлении в разделе переменных) может быть произвольным. Использование указателей, которым не присвоено значение процедурой *New* или другим способом, не контролируется Delphi и вызовет исключение.

Как уже отмечалось, параметром процедуры *New* может быть только типизированный указатель. Для работы с нетипизированными указателями используются процедуры *GetMem* и *FreeMem*:

```
GetMem(P, Size); // Резервирование памяти
FreeMem(P, Size); // Освобождение памяти
```

Здесь *P* — нетипизированный указатель; *Size* — размер в байтах требуемой или освобождаемой части кучи.

СОВЕТ Использование процедур *GetMem* и *FreeMem*, как и вообще вся работа с динамической памятью, требует особой осторожности и тщательного соблюдения простого правила: освобождать нужно ровно столько памяти, сколько ее было зарезервировано, и именно с того адреса, с которого она была зарезервирована.

В табл. 6.10 приводится описание как уже рассмотренных процедур и функций Delphi, так и некоторых других, которые могут оказаться полезными при обращении к динамической памяти.

Таблица 6.10. Средства Delphi для работы с памятью

Подпрограмма	Описание
function Addr(X): Pointer;	Возвращает адрес аргумента X. Аналогичный результат возвращает операция @
procedure Dispose(var P: Pointer);	Возвращает в кучу фрагмент динамической памяти, который ранее был зарезервирован за типизированным указателем P

продолжение ➤

Таблица 6.10 (продолжение)

Подпрограмма	Описание
procedure FreeMem(var P: Pointer ; Size: Integer);	Возвращает в кучу фрагмент динамической памяти, который ранее был зарезервирован за нетипизированным указателем P
procedure GetMem(var P: Pointer ; Size: Integer);	Резервирует за нетипизированным указателем P фрагмент динамической памяти требуемого размера Size
procedure New(var P: Pointer);	Резервирует фрагмент кучи для размещения переменной и помещает в типизированный указатель P адрес первого байта
function SizeOf(X): Integer;	Возвращает длину в байтах внутреннего представления указанного объекта X

Windows имеет собственные средства работы с памятью. За более полной информацией обращайтесь к справочной службе в файле WIN32.HLP или WIN32S.HLP.

Псевдонимы типов

Для любого типа можно объявить сколько угодно псевдонимов. Например:

```
type
  TMyInteger = Integer;
```

В дальнейшем псевдоним можно использовать так же, как и базовый тип:

```
var
  MyInt: TMyInteger;
begin
  MyInt := 2*Round(pi);
  ...
end;
```

Такого рода псевдонимы обычно используются для лучшей наглядности кода программы. Однако в Delphi можно объявлять строго типизированные псевдонимы добавлением зарезервированного слова **type** перед именем базового типа:

```
type
  TMyIntegerType = type Integer;
var
  MyIntVar: TMyIntegerType;
```

С точки зрения компилятора, типизированные псевдонимы совместимы с базовым типом в различного рода выражениях, но фактически они объявляют *новый тип данных*, поэтому их нельзя использовать в качестве формальных параметров обращения к подпрограммам вместо базового типа. Пусть, например, объявлена процедура:

```
function MyIntFunc(APar: Integer): Integer;
begin
  ...
end;
```

Тогда следующее обращение к ней будет расценено компилятором как ошибочное:

```
MyIntFunc(MyIntVar)
```

Строго типизированные псевдонимы заставляют компилятор вырабатывать информацию о типе для этапа прогона программы (Run-Time Type Information, RTTI). Эта информация обычно используется средой Delphi, чтобы обеспечить функционирование разного рода редакторов свойств и программ-экспертов (мастеров).

Типы с управляемым временем жизни

Некоторые типы (**String**, **WideString**, **Variant**, **OleVariant**, **interface** и **dispinterface**) являются типами *с управляемым временем жизни*. Особенностью такого рода типов является то, что выделенная для размещения переменной область памяти автоматически возвращается системе после того, как переменная перестает существовать (например, локальные переменные в подпрограммах). Освобождение выделенных ресурсов часто называется «сборкой мусора» (garbage-collected).

Освобождение памяти, выделенной для глобальных переменных, обычно реализуется в ходе завершения работы программы. Для локальных подпрограмм этот процесс приводит к некоторому их усложнению. Пусть, например, имеем такую процедуру:

```
procedure MyProc;  
var  
  S: String;  
begin  
  // В теле процедуры используется переменная S  
end;
```

В действительности компилятор вырабатывает код, соответствующий такому тексту (о блоках **try** и **finally** см. главу 13.):

```
procedure MyProc;  
var  
  S: String;  
begin  
  S := '';  
  try  
    // В теле процедуры используется переменная S  
  finally  
    // Здесь реализуется освобождение памяти для S  
  end;
```


Глава 7

Процедуры и функции

Процедуры и функции представляют собой относительно самостоятельные фрагменты программы, оформленные особым образом и снабженные именем. Упоминание этого имени в тексте программы называется *вызовом* процедуры (функции). Отличие функции от процедуры заключается в том, что результатом исполнения операторов, образующих тело функции, всегда является некоторое значение, поэтому обращение к функции можно использовать в соответствующих выражениях наряду с переменными и константами. Условимся далее называть процедуру или функцию общим именем «подпрограмма», если только для излагаемого материала указанное отличие не имеет значения.

Локализация имен

Напомним, что вызов подпрограммы осуществляется простым упоминанием имени процедуры в операторе вызова процедуры или имени функции в выражении. В 32-разрядной версии Delphi функцию можно вызывать точно так же, как и процедуру, то есть без использования возвращаемого ею значения. Для этого следует задействовать так называемый расширенный синтаксис языка, который вводится установкой флажка Extended Syntax на вкладке Compiler окна Options (вызывается командой Project ► Options) или заданием директивы компилятора `{SX+}`.

Как уже отмечалось, любое имя в программе должно быть обязательно описано, перед тем как оно появится среди исполняемых операторов. Не делается исключения и в отношении подпрограмм: каждую свою процедуру и функцию программисту необходимо описать в разделе описаний.

Описать подпрограмму — значит указать ее заголовок и тело. В заголовке объявляются имя подпрограммы и формальные параметры, если они есть. Для функции, кроме того, указывается тип возвращаемого ею результата. За заголовком следует тело подпрограммы, которое, подобно программе, состоит из раздела описаний и раздела исполняемых операторов. В разделе описаний подпрограммы могут встретиться описания подпрограмм низшего уровня, а в них — описания других подпрограмм и т. д.

Вот какую иерархию описаний получим, например, для программы, структура которой представлена на рис. 7.1 (для простоты считается, что все подпрограммы представляют собой процедуры без параметров):

```

procedure A;
  procedure A1;
  ...
  begin
  ...
  end {A1};
procedure A2;
  ...
  begin
  ...
  end {A2};
begin {A}
  ...
end {A};
procedure B;
  procedure B1;
  ...
  begin
  ...
  end {B1};
procedure B2;
  procedure B21;
  ...

```

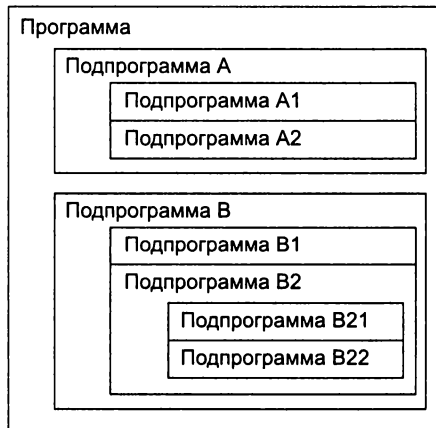


Рис. 7.1. Пример структуры программы

Подпрограмма любого уровня имеет обычно множество имен констант, переменных, типов и вложенных в нее подпрограмм низшего уровня. Считается, что все имена, описанные внутри подпрограммы, локализируются в ней, то есть они как

бы «невидимы» снаружи подпрограммы. Таким образом, для операторов, обращающихся к подпрограмме, она представляется как «черный ящик», в котором реализуется тот или иной алгоритм. Все детали этой реализации скрыты от глаз пользователя подпрограммы и потому недоступны ему. Например, в рассмотренном выше примере из основной программы можно обратиться к процедурам А и В, но нельзя вызвать ни одну из вложенных в них процедур А1, А2, В1 и т. д.

Сказанное относится не только к именам самих подпрограмм, но и вообще к любым объявленным в них именам — типам, константам, переменным и меткам. Все имена в пределах подпрограммы, в которой они объявлены, должны быть уникальными и не могут совпадать с именем самой подпрограммы.

При входе в подпрограмму низшего уровня становятся доступными не только объявленные в ней имена, но и сохраняется доступ ко всем именам верхнего уровня. Образно говоря, любая подпрограмма как бы окружена полупрозрачными стенками: снаружи подпрограммы мы не видим ее внутренности, но, попав в подпрограмму, можем наблюдать все, что делается снаружи. Так, например, из подпрограммы В21 мы можем вызвать подпрограмму А, использовать имена, объявленные в основной программе, в подпрограммах В и В2, и даже обратиться к ним. Любая подпрограмма может, наконец, вызвать саму себя — такой способ вызова называется *рекурсией*.

Пусть имеем такое описание:

```
var V1 : ...;
  procedure A;
  var V2 : ...;
  ...
end {A};
  procedure B;
  var V3 : ...;
  procedure B1;
  var V4 : ...;
  procedure B11;
  var V5;
  ...
```

Из процедуры В11 доступны все пять переменных V1, ..., V5, из процедуры В1 доступны переменные V1, ..., V4, из центральной программы — только V1.

При взаимодействии подпрограмм одного уровня иерархии вступает в силу основное правило Delphi: любая подпрограмма перед ее использованием должна быть описана. Поэтому из подпрограммы В можно вызвать подпрограмму А, но из А вызвать В невозможно, точнее, такая возможность появляется только с использованием опережающего описания (см. ниже раздел «Рекурсия и опережающее описание»). Продолжая образное сравнение, подпрограмму можно уподобить ящику с непрозрачными стенками и дном, но полупрозрачной крышей: из подпрограммы можно смотреть только «вверх» и нельзя «вниз», то есть подпрограмме доступны только те объекты верхнего уровня, которые описаны до описания данной подпрограммы. Эти объекты называются *глобальными* по отношению к подпрограмме.

В Delphi допускается произвольная последовательность описания констант, переменных, типов, меток и подпрограмм. Например, раздел **var** описания переменных может появляться в пределах раздела описаний одной и той же подпрограммы много раз и перемежаться с объявлениями других объектов и подпрограмм. Для Delphi совершенно безразличен порядок следования и количество разделов **var**, **type**, **const** и **label**, но при определении области действия этих описаний следует помнить, что имена, описанные ниже по тексту программы, недоступны из ранее описанных подпрограмм, например:

```
var V1 : ...;
  procedure S;
    var V2 : ...;
    ...
  end {S};
var V3 : ...;
...
```

Из процедуры *S* можно обратиться к переменным *V1* и *V2*, но нельзя использовать *V3*, так как описание этой переменной следует в программе за описанием процедуры *S*.

Локализованные в подпрограмме имена могут совпадать с ранее объявленными глобальными именами. В этом случае считается, что локальное имя «закрывает» глобальное и делает его недоступным, например:

```
var
  i : Integer;
  procedure P;
    var
      i : Integer;
    begin
      lbOutput.Caption := IntToStr(i);
    end {P};
begin
  i := 1;
  P
end;
```

Что выведет эта программа на экран? Все, что угодно: значение внутренней переменной *i* при входе в процедуру *P* не определено, хотя одноименная глобальная переменная имеет значение 1. Локальная переменная «закрывает» глобальную, и на экран будет выведено произвольное значение, содержащееся в неинициализированной внутренней переменной. Если же убрать из процедуры *P* показанное ниже описание, то на экран будет выведено значение глобальной переменной *i*, то есть 1:

```
var
  i : integer;
```

Таким образом, одноименные глобальные и локальные переменные — это разные переменные. Любое обращение к таким переменным в теле подпрограммы трактуется как обращение к локальным переменным, то есть глобальные переменные в этом случае попросту недоступны.

Описание подпрограммы

Описание подпрограммы состоит из заголовка и тела подпрограммы.

Заголовок и стандартные директивы

Заголовок процедуры имеет вид:

```
procedure <имя> [ (<сп.ф.п.>) ] ;
```

Заголовок функции:

```
function <имя> [ (<сп.ф.п.>) ] : <тип>;
```

Здесь <имя> — имя подпрограммы (правильный идентификатор); <сп.ф.п.> — список формальных параметров; <тип> — тип возвращаемого функцией результата.

Сразу за заголовком подпрограммы может следовать одна из стандартных директив **assembler**, **external**, **far**, **forward**, **inline**, **interrupt**, **near**. Эти директивы уточняют действия компилятора и распространяются на всю подпрограмму и только на нее, то есть если за подпрограммой следует другая подпрограмма, стандартная директива, указанная за заголовком первой, не распространяется на вторую.

- **assembler** — эта директива отменяет стандартную последовательность машинных инструкций, вырабатываемых при входе в процедуру и перед выходом из нее. Тело подпрограммы в этом случае должно реализоваться с помощью команд встроенного ассемблера.
- **external** — с помощью этой директивы объявляется внешняя подпрограмма.
- **far** — компилятор должен создавать код подпрограммы, рассчитанный на дальнюю модель вызова. Директива **near** заставит компилятор создать код, рассчитанный на ближнюю модель памяти. Обе директивы введены для совместимости с Delphi версии 1, в которой использовалась сегментная модель памяти.
- **forward** — используется при опережающем описании (см. раздел «Рекурсия и опережающее описание») для сообщения компилятору, что описание подпрограммы следует где-то дальше по тексту программы (но в пределах текущего программного модуля).
- **inline** — указывает на то, что тело подпрограммы реализуется с помощью встроенных машинных инструкций.
- **interrupt** — используется при создании процедур обработки прерываний.

Помимо описанных в Delphi можно задействовать также стандартные директивы, регламентирующие способ передачи параметров через стек и применение регистров для их передачи — такие директивы используются при работе с ядром Windows (табл. 7.1). Столбец *Порядок* определяет порядок размещения параметров в стеке: *Слева направо* означает размещение в стеке по порядку описания — сначала первый параметр, затем второй и т. д.; *Справа налево* означает размещение с конца перечисления параметров — сначала последний, затем предпоследний и т. д. Столбец *Очистка* определяет, кто будет очищать стек: подпрограмма перед передачей управления в вызывающую программу, или программа после получения

управления. В столбце *Регистры* указывается *Да*, если для передачи параметров помимо стека используются также регистры центрального процессора, и *Нет* в противном случае.

Таблица 7.1. Стандартные директивы, регламентирующие способ передачи параметров через стек и использование регистров для их передачи

Директива	Порядок	Очистка	Регистры
register	Слева направо	Подпрограмма	Да
pascal	Слева направо	Подпрограмма	Нет
cdecl	Справа налево	Программа	Нет
stdcall	Справа налево	Подпрограмма	Нет
safecall	Справа налево	Подпрограмма	Нет

ВНИМАНИЕ

Для поддержки всех функций API ядра Windows подпрограмма должна компилироваться с директивой **stdcall**, а для поддержки функций COM — с директивой **safecall**.

Параметры

Список формальных параметров необязателен и может отсутствовать. Если же он есть, то в нем должны быть перечислены имена формальных параметров и их типы, например:

```
procedure SB(a: Real; b: Integer; c: Char);
```

Как видно из примера, параметры в списке отделяются друг от друга точкой с запятой. Несколько следующих подряд однотипных параметров можно объединять в подписки, например, следующие два описания эквивалентны:

```
function F(a: Real; b: Real): Real;  
function F(a,b: Real): Real;
```

Для операторов тела подпрограммы список формальных параметров является своеобразным расширением раздела описаний: все переменные из этого списка могут использоваться в любых выражениях внутри подпрограммы. Таким способом осуществляется настройка алгоритма подпрограммы на конкретную задачу.

ВНИМАНИЕ

В отличие от процедуры, любая функция имеет внутреннюю переменную *Result* того же типа, что и тип функции. Присваивание этой переменной значения трактуется как указание результата, возвращаемого функцией. В стандартном языке Паскаль такой переменной нет, и для указания возвращаемого результата нужно присваивать значение *названию* функции. Например:

```
function MyFunc: Integer;  
begin  
    MyFunc := 0  
end;
```

ПРИМЕЧАНИЕ В Delphi упоминание имени функции в левой части оператора присваивания означает рекурсивный вызов функции (см. ниже раздел «Рекурсия и опережающее описание»). Следует писать так:

```
function MyFunc: Integer;
begin
    Result := 0
end;
```

Рассмотрим такой полезный пример. В Delphi не предусмотрена операция возведения вещественного числа в произвольную степень¹. Тем не менее эту задачу можно решить с использованием стандартных математических функций Exp и Ln по следующему алгоритму:

$$X^Y = e^{(Y \times \ln(X))}$$

Создадим функцию с именем Power и двумя вещественными параметрами A и B, которая будет возвращать результат возведения A в степень B. Обработчик события bbRunClick нашей учебной формы fmExample читает из компонента edInput текст и пытается выделить из него два числа, разделенных хотя бы одним пробелом. Если это удалось сделать, он обращается к функции Power дважды: сначала возводит первое число X в степень второго числа Y, затем X возводится в степень -Y (проект Source\Chap_07\Power\Power.dpr).

```
procedure TfmExample.bbRunClick(Sender: TObject);
function Power(A, B: Real): Real;
{ Функция возводит число A в степень B. Поскольку логарифм отрицательного числа не существует, реализуется проверка значения A: отрицательное значение заменяется положительным, для нулевого числа результат равен нулю. Кроме того, любое число в нулевой степени дает единицу. }
begin
    if A > 0 then
        Result := Exp(B * Ln(A))
    else if A < 0 then
        Result := Exp(B * Ln(Abs(A)))
    else if B = 0 then
        Result := 1
    else
        Result := 0;
end;           // Power
var
    S: String;
    X, Y: Real;
begin
```

¹ Начиная с версии 2, с Delphi поставляется модуль Math, в котором соответствующая функция есть (см. приложение Б).

```

{ Читаем строку из edInput и выделяем из нее два
вещественных числа, разделенных хотя бы одним пробелом }
S := edInput.Text;
if (S = '') or (pos(' ', S) = 0) then
  Exit; // Нет текста или в нем нет
        // пробела - прекращаем дальнейшую работу
try
  // Выделяем первое число:
  X := StrToFloat(copy(S, 1, pos(' ', S) - 1));
  // Если успешно, удаляем символы до пробела
  // и выделяем второе число:
  Delete(S, 1, pos(' ', S));
  Y := StrToFloat(Trim(S));
except
  Exit; // Завершаем работу при ошибке преобразования
end;
mmOutput.Lines.Add('X в степени Y =
'+FloatToStr(Power(X, Y)));
mmOutput.Lines.Add('X в степени -Y =
'+FloatToStr(Power(X, -Y)));
end;

```

Для вызова функции `Power` мы просто указали ее в качестве параметра при обращении к стандартной функции преобразования вещественного числа в строку `FloatToStr`. Параметры `X` и `Y` в момент обращения к функции `Power` — это фактические параметры. Они подставляются вместо формальных параметров `A` и `B` в заголовке функции и затем над ними осуществляются нужные действия. Полученный результат присваивается специальной переменной с именем `Result`, которая в теле любой функции интерпретируется как то значение, которое вернет функция после окончания своей работы. В программе функция `Power` вызывается дважды — сначала с параметрами `X` и `Y`, затем с параметрами `X` и `-Y`, поэтому будут получены два разных результата.

Механизм замены формальных параметров фактическими позволяет нужным образом настроить алгоритм, реализованный в подпрограмме. Delphi следит за тем, чтобы количество и типы формальных параметров строго соответствовали количеству и типам фактических параметров в момент обращения к подпрограмме. Смысл используемых фактических параметров зависит от того, в каком порядке они перечислены при вызове подпрограммы. В нашем примере первый по порядку фактический параметр возводится в степень, задаваемую вторым параметром, а не наоборот. Программист должен сам следить за правильным порядком перечисления фактических параметров при обращении к подпрограмме.

Любой из формальных параметров подпрограммы может быть либо параметром-значением, либо параметром-переменной, либо, наконец, параметром-константой.

В предыдущем примере параметры `A` и `B` определены как параметры-значения. Если параметры определяются как параметры-переменные, перед ними необходимо

ставить зарезервированное слово **var**, а если это параметры-константы, — слово **const**, например:

```
procedure Myprocedure (var A: Real; B: Real; const C:
    String);
```

Здесь А — параметр-переменная, В — параметр-значение, а С — параметр-константа.

Определение формального параметра тем или иным способом существенно, в основном, только для вызывающей программы: если формальный параметр объявлен как параметр-переменная, то при вызове подпрограммы ему должен соответствовать фактический параметр в виде *переменной* нужного типа; если формальный параметр объявлен как параметр-значение или параметр-константа, то при вызове ему может соответствовать произвольное *выражение*. Контроль за неукоснительным соблюдением этого правила осуществляется компилятором Delphi. Предположим, для предыдущего примера был использован такой заголовок функции:

```
function Power (A: Real; var B : Real): Real;
```

Тогда при втором обращении к функции компилятор указал бы на несоответствие типа фактических и формальных параметров (параметр обращения — Y есть выражение, в то время как соответствующий ему формальный параметр В описан как параметр-переменная).

Для того чтобы понять, в каких случаях использовать тот или иной тип параметра, рассмотрим, как осуществляется замена формальных параметров фактическими в момент обращения к подпрограмме.

Если параметр определен как параметр-значение, то перед вызовом подпрограммы это значение вычисляется, полученный результат копируется во временную память (стек) и передается подпрограмме. Важно учесть, что даже если в качестве фактического параметра указано простейшее выражение в виде переменной или константы, все равно подпрограмме будет передана лишь копия переменной (константы). Любые возможные изменения в подпрограмме параметра-значения никак не воспринимаются вызывающей программой, так как в этом случае изменяется копия фактического параметра.

Если параметр определен как параметр-переменная, то при вызове подпрограммы передается сама переменная, а не ее копия (фактически в этом случае подпрограмме передается *адрес* переменной). Изменение параметра-переменной приводит к изменению фактического параметра в вызывающей программе.

В случае параметра-константы в подпрограмму также передается адрес области памяти, в которой располагается переменная или вычисленное значение. Однако компилятор блокирует любые присваивания параметру-константе нового значения в теле подпрограммы.

Итак, параметры-переменные используются как средство связи алгоритма, реализованного в подпрограмме, с внешним миром: с помощью этих параметров подпрограмма может передавать результаты своей работы вызывающей программе. Разумеется, в распоряжении программиста всегда есть и другой способ передачи результатов — через глобальные переменные. Однако злоупотребление глобальными связями делает программу, как правило, запутанной, трудной в понимании и сложной в отладке. В соответствии с требованиями хорошего стиля программи-

рования рекомендуется там, где это возможно, передавать результаты через фактические параметры-переменные.

С другой стороны, описание всех формальных параметров как параметров-переменных нежелательно по двум причинам. Во-первых, это исключает возможность вызова подпрограммы с фактическими параметрами в виде выражений, что делает программу менее компактной. Во-вторых, и это главное, в подпрограмме возможно случайное использование формального параметра, например, для временного хранения промежуточного результата, то есть всегда существует опасность непреднамеренно испортить фактическую переменную. Вот почему параметрами-переменными следует объявлять только те, через которые подпрограмма в действительности передает результаты вызывающей программе. Чем меньше параметров объявлено параметрами-переменными и чем меньше в подпрограмме используется глобальных переменных, тем меньше опасность получения непредусмотренных программистом побочных эффектов, связанных с вызовом подпрограммы, тем проще программа в понимании и отладке. По той же причине не рекомендуется использовать параметры-переменные в заголовке функции: если результатом работы функции не может быть единственное значение, то логичнее задействовать процедуру или нужным образом декомпозировать алгоритм на несколько подпрограмм.

Для передачи результатов вызывающей программе можно использовать параметры-переменные, которым предшествует зарезервированное слово **out**. Например:

```
procedure AProc(out A: Real; B: Integer);
```

В этом примере параметр A есть параметр-переменная, поэтому вызов процедуры может быть, например, таким:

```
var
  X: Real;
begin
  AProc(X, 123);
  ...
end;
```

Параметры **out** нельзя использовать в подпрограмме как входные, они предназначены исключительно для передачи результата вызывающей программе.

Существует еще одно обстоятельство, которое следует учитывать при выборе вида формальных параметров. Как уже говорилось, при объявлении параметра-значения осуществляется копирование фактического параметра во временную память. Если этим параметром будет массив большой размерности, то существенные затраты времени и памяти на копирование при многократных обращениях к подпрограмме можно минимизировать, объявив этот параметр параметром-константой. Параметр-константа не копируется во временную область памяти, что сокращает затраты времени на вызов подпрограммы, а любые его изменения в теле подпрограммы невозможны — за этим строго следит компилятор.

Еще одно свойство Delphi — возможность использования *нетипизированных параметров*. Параметр считается нетипизированным, если тип формального параметра-переменной в заголовке подпрограммы не указан, при этом соответствующий

ший ему фактический параметр может быть переменной любого типа. Заметим, что нетипизированными могут быть только параметры-переменные:

```
procedure MyProc (var aParametr);
```

Нетипизированные параметры обычно используются в случае, когда тип данных несущественен. Такие ситуации чаще всего возникают при копировании одной области памяти в другую, например, с помощью процедур `BlockRead`, `BlockWrite`, `MoveMemory` и т. п.

Как уже говорилось, список параметров подпрограммы может отсутствовать. В этом случае по правилам стандартного языка Паскаль опускаются и обрамляющие список круглые скобки. Однако в Delphi 7 скобки можно оставить:

```
procedure MyProc ();
```

Это сделано для упрощения перехода на Delphi программистов, активно использующих языки C, C++, Java.

Умалчиваемые параметры

В Delphi можно использовать так называемые умалчиваемые параметры, то есть параметры, которые могут опускаться при обращении к подпрограмме. Умалчиваемые параметры замыкают список формальных параметров и имеют вид:

```
<имя>: <тип> = <значение>
```

Например:

```
procedure P(a: array of Integer; S: String = '');
```

В этом случае два следующих обращения идентичны:

```
P([1, 2, 3], '');
```

```
P([1, 2, 3]);
```

Если в подпрограмме используются два и более умалчиваемых параметра, то в случае переопределения одного из них при обращении к подпрограмме следует указывать все параметры вплоть до последнего переопределяемого (то есть нельзя заменять запятыми те умалчиваемые параметры, которые не переопределяются). Например:

```
procedure P(a: Integer; S: String = ''; B: Integer = 0);
```

Допустимые обращения:

```
P(123);
```

```
P(123, 'Строка');
```

```
P(123, '', 1)
```

Умалчиваемые параметры должны следовать в конце списка параметров. Они могут быть только простых типов и передаются по значению или с модификатором **const**.

Параметры-массивы и параметры-строки

Может сложиться впечатление, что объявление переменных в списке формальных параметров подпрограммы ничем не отличается от объявления их в разделе описания переменных. Действительно, в обоих случаях много общего, но есть одно

существенное различие: типом любого параметра в списке формальных параметров может быть только стандартный или ранее объявленный тип. Поэтому нельзя, например, объявить следующую процедуру:

```
procedure S (a: array [1..10] of real);
```

Причина в том, что в списке формальных параметров фактически объявляется тип-диапазон, указывающий границы индексов массива.

Если мы хотим передать какой-то элемент массива, то проблем, как правило, не возникает, но если в подпрограмму передается весь массив, то следует первоначально описать его тип. Например:

```
type
  aType = array [1..10] of Real;
procedure S(var a: aType);
...
```

Поскольку короткая строка является фактически своеобразным массивом, ее передача в подпрограмму осуществляется аналогичным образом:

```
type
  InputType = String [15];
  OutputType = String [30];
function St(S: InputType): OutputType;
...
```

Требование описать любой тип-массив или тип-строку перед объявлением подпрограммы, на первый взгляд, кажется несущественным. Действительно, в рамках простейших вычислительных задач обычно заранее известна структура всех используемых в программе данных, поэтому статическое описание массивов не вызывает проблем. Однако разработка программных средств универсального назначения связана со значительными трудностями.

Открытые массивы

Delphi поддерживает так называемые *открытые массивы*, легко решающие проблему передачи подпрограмме одномерных массивов переменной длины.

Открытый массив представляет собой формальный параметр подпрограммы, описывающий базовый тип элементов массива, но не определяющий его размерности и границы:

```
procedure MyProc (OpenArray: array of Integer);
```

Внутри подпрограммы такой параметр трактуется как одномерный массив с нулевой нижней границей. Верхняя граница открытого массива возвращается стандартной функцией `High`. Используя 0 как минимальный индекс и значение, возвращаемое функцией `High`, как максимальный индекс, подпрограмма может обрабатывать одномерные массивы произвольной длины.

```
procedure TfmExample.bbRunClick(Sender: TObject);
{Иллюстрация использования открытых массивов: программа
выводит в компонент tmOutput содержимое двух одномерных
массивов разной длины с помощью одной процедуры ArrayPrint}
```

```

procedure ArrayPrint(aArray: array of Integer);
var
    k: Integer;
    S: String;
begin
    S := '';
    for k := 0 to High(aArray) do
        S := S + IntToStr(aArray[k]);
    mmOutput.Lines.Add(S);
end;
const
    A: array [-1..2] of Integer = (0,1,2,3);
    B: array [5..7] of Integer = (4,5,6);
begin
    ArrayPrint(A);
    ArrayPrint(B);
end;

```

Как видно из этого примера, фактические границы массивов A и B, передаваемых в качестве параметров вызова процедуре `ArrayPrint`, не имеют значения. Однако размерность открытых массивов (количество индексов) всегда равна 1 — за этим следит компилятор. Например, добавим в программу двухмерный массив C:

```

var
    C: array [1..3,1..5] of Integer;

```

Тогда следующее обращение вызовет сообщение об ошибке.

```
ArrayPrint(C)
```

Delphi поддерживает также открытый массив констант, который задается ключевыми словами **array of const**. Такой массив может содержать данные различных типов. Например:

```

procedure MyProc(array of const);
...
MyProc(['', 123, @MyProc, 3.14, False]);

```

Конструктор массива

При обращении к подпрограмме на месте формального параметра в виде открытого массива можно указывать так называемый конструктор массива. Конструктор массива представляет собой список разделенных запятыми значений элементов массива, обрамленный квадратными скобками. Вспомним предыдущий пример:

```

const
    A: array [-1..2] of Integer = (0,1,2,3);
    B: array [5..7] of Integer = (4,5,6);
begin
    ArrayPrint(A);
    ArrayPrint(B);

```

```
end;
```

Обращения к массивам в этом примере мы могли бы написать так:

```
begin
```

```
    ArrayPrint([0,1,2,3]);
```

```
    ArrayPrint([4,5,6]);
```

```
end;
```

Вариантные массивы-параметры

В Delphi при передаче подпрограмме массивов переменной длины и размерности удобно использовать варианты массивы (подробнее о вариантах см. главу 9). В следующем примере с помощью функции `GetArrayAverage` определяется среднее арифметическое значение всех элементов вариантного массива произвольной длины и размерности не выше 5:

```
function GetArrayAverage(const V: Variant): Double;
```

```
{Возвращает среднее арифметическое значение массива произвольной длины и размерности или очень маленькую отрицательную величину, если V - не вариант или если его размерность больше 5}
```

```
var
```

```
    i,j,k,l,m: Integer;
```

```
    Sum: Double;
```

```
    Nitem: Integer;
```

```
begin
```

```
    Result := -1E-309;
```

```
    if ((VarType(V) and VarArray) <> VarArray) or  
        (VarArrayDimCount(V) > 5) then
```

```
        Exit;
```

```
    Sum := 0;
```

```
    Nitem := 0;
```

```
    // Подсчитываем количество элементов массива
```

```
    for k := 1 to VarArrayDimCount(V) do
```

```
        Nitem := Nitem+VarArrayHighBound(V,k) -
```

```
VarArrayLowBound(V,k);
```

```
    // Подсчитываем сумму элементов
```

```
    case VarArrayDimCount(V) of
```

```
1: for i:=VarArrayLowBound(V,1) to VarArrayHighBound(V,1) do  
        Sum := Sum+V[i];
```

```
2: for i:=VarArrayLowBound(V,1) to VarArrayHighBound(V,1) do  
        for j:=VarArrayLowBound(V,2) to VarArrayHighBound(V,2) do  
            Sum := Sum+V[i,j];
```

```
3: for i:=VarArrayLowBound(V,1) to VarArrayHighBound(V,1) do  
        for j:=VarArrayLowBound(V,2) to VarArrayHighBound(V,2) do  
            for k:=VarArrayLowBound(V,3) to VarArrayHighBound(V,3) do  
                Sum := Sum+V[i,j,k];
```

```
4: for i:=VarArrayLowBound(V,1) to VarArrayHighBound(V,1) do
```

```

    for j:=VarArrayLowBound(V,2) to VarArrayHighBound(V,2) do
    for k:=VarArrayLowBound(V,3) to VarArrayHighBound(V,3) do
    for l:=VarArrayLowBound(V,4) to VarArrayHighBound(V,4) do
        Sum := Sum+V[i,j,k,l];
5: for i:=VarArrayLowBound(V,1) to VarArrayHighBound(V,1) do
    for j:=VarArrayLowBound(V,2) to VarArrayHighBound(V,2) do
    for k:=VarArrayLowBound(V,3) to VarArrayHighBound(V,3) do
    for l:=VarArrayLowBound(V,4) to VarArrayHighBound(V,4) do
    for m:=VarArrayLowBound(V,5) to VarArrayHighBound(V,5) do
        Sum := Sum+V[i,j,k,l,m];
end;
Result := Sum/NItem
end;
```

Как нетрудно догадаться, функция `VarArrayDimCount()` возвращает размерность вариантного массива.

Процедурные типы

Основное назначение процедурных типов — дать программисту гибкие средства передачи функций и процедур в качестве фактических параметров обращения к другим процедурам и функциям.

Для объявления процедурного типа используется заголовок процедуры (функции), в котором опускается ее имя, например:

```

type
  Proc1 = procedure (a, b, c: Real; var d: Real);
  Proc2 = procedure (var a, b);
  Proc3 = procedure ();
  Func1 = function: String;
  Func2 = function (var s: String): Real;
```

Как видно из приведенных примеров, существует два процедурных типа: тип-процедура и тип-функция.

В следующей программе (проект `Source\Chap_07\ProcType\ProcType.dpr`) иллюстрируется механизм передачи процедур в качестве фактических параметров вызова. Программа выводит на экран таблицу двух функций:

$$\begin{aligned} \text{Sin1}(x) &= (\text{Sin}(x) + 1) \times \text{Exp}(-x) \\ \text{Cos1}(x) &= (\text{Cos}(x) + 1) \times \text{Exp}(-x) \end{aligned}$$

Вычисление и печать значений этих функций реализуются в процедуре `PrintFunc`, которой в качестве параметров передается количество NP вычислений функции в диапазоне X от 0 до $2 \times \pi$ и имя нужной функции.

```

function Sin1(X: Real): Real;
begin
    Result := (Sin(X) + 1) * Exp(-X)
end;           // Sin1

function Cos1(X: Real): Real;
```

```

begin
  Result := (Cos(X) + 1) * Exp(-X)
end;          // Cos1

procedure TfmExample.bbRunClick(Sender: TObject);
type
  Func = function (X: Real): Real;          // Процедурный тип
procedure PrintFunc(NP: Integer; F: Func) ;
var
  k: Integer;
  X: Real;
begin
  for k := 0 to NP do
    begin
      X := k * 2 * pi / NP;
      mmOutput.Lines.Add(FloatToStrF(X, ffExponent, 10, 2) +
        #9#9 + FloatToStrF(F(X), ffExponent, 10, 2));
    end;
  end;          // PrintFunc

begin          // bbRunClick
  mmOutput.Lines.Add(#9'Функция SIN1:');
  PrintFunc(10, Sin1);
  mmOutput.Lines.Add(#9'Функция COS1:');
  PrintFunc(10, Cos1);
end;

```

Обратите внимание: передаваемые подпрограммы не могут быть локальными, то есть процедурами или функциями, объявленными внутри другой подпрограммы. Вот почему описание подпрограмм Sin1 и Cos1 размещаются вне обработчика bbRunClick, но выше него по тексту модуля. Замечу, что символ #9 — это символ табуляции, который вставляется в формируемые строки для разделения колонок с цифрами.

В программе могут быть объявлены переменные процедурных типов, например, так:

```

var
  p1      : Proc1;
  f1, f2: Func2;
  ap      : array [1..N] of Proc1;

```

Переменным процедурных типов допускается присваивать в качестве значений имена соответствующих подпрограмм. После такого присваивания имя переменной становится синонимом имени подпрограммы.

Рекурсия и опережающее описание

Рекурсия — это такой способ организации вычислительного процесса, при котором подпрограмма в ходе выполнения составляющих ее операторов обращается сама к себе.

Рассмотрим классический пример — вычисление факториала. Программа получает от компонента `edInput` целое число `N` и выводит в компонент `mmOutput` значение $N!$, которое вычисляется с помощью рекурсивной функции `Factorial`.

При выполнении правильно организованной рекурсивной подпрограммы осуществляется многократный переход от некоторого текущего уровня организации алгоритма к нижнему уровню последовательно до тех пор, пока, наконец, не будет получено тривиальное решение поставленной задачи. В нашем случае решение при $N=0$ тривиально (факториал 0 равен 1) и используется для остановки рекурсии (проект `Source\Chap_07\Factorial\Factorial.dpr`).

```
procedure TfmExample.bbRunClick(Sender: TObject);
function Factorial(N: Word): Extended;
begin
    if N = 0 then
        Result := 1
    else
        Result := N * Factorial(N-1)
end; // Factorial
var
    N: Integer;
begin
    try
        N := StrToInt(Trim(edInput.Text));
    except
        Exit;
    end;
    mmOutput.Lines.Add(edInput.Text+
        '!' = '+FloatToStr(Factorial(N))');
    edInput.Text := '';
    edInput.SetFocus
end;
```

Рекурсивная форма организации алгоритма обычно выглядит изящнее итерационной и дает более компактный текст программы, но при выполнении, как правило, медленнее и может вызвать переполнение стека (при каждом входе в подпрограмму ее локальные переменные размещаются в организованной особым образом области памяти, называемой программным стеком).

Рекурсивный вызов может быть косвенным. В этом случае подпрограмма обращается к себе опосредованно, путем вызова другой подпрограммы, в которой содержится обращение к первой, например:

```
procedure A (i : Byte);
begin
    ...
    B (i);
    ...
end;
procedure B (j : Byte);
```

```
...  
begin  
  ...  
  A(j);  
  ...  
end;
```

Если строго следовать правилу, согласно которому каждый идентификатор перед употреблением должен быть описан, то такую программную конструкцию использовать нельзя. Чтобы такого рода вызовы стали возможны, вводится *опережающее описание*:

```
procedure B (j : Byte); Forward;  
procedure A (i : Byte);  
begin  
  ...  
  B (i);  
  ...  
end;  
procedure B;  
begin  
  ...  
  A(j);  
  ...  
end;
```

Как видим, опережающее описание заключается в том, что объявляется лишь заголовок процедуры В, а ее тело заменяется стандартной директивой **Forward**. Теперь в процедуре А можно использовать обращение к процедуре В — ведь она уже описана, точнее, известны ее формальные параметры, и компилятор может правильным образом организовать ее вызов. Обратите внимание: тело процедуры В начинается заголовком, в котором уже не указываются описанные ранее формальные параметры.

Глава 8

Классы

Классами в Delphi называются специальные типы, которые содержат поля, методы и свойства. Как и любой другой тип, класс служит лишь образцом для создания конкретных экземпляров реализации, которые называются объектами. Сразу же уточню, что в предшественнице Delphi — системе программирования Turbo Pascal — объектами называются типы, имеющие много общего с классами Delphi. Однако существенные усовершенствования, внесенные в объектную модель Delphi, заставили разработчиков языка ввести для обозначения объектов специальный термин — *класс*, заимствованный, кстати, из C++. Для совместимости с ранее разработанными программами системы Turbo Pascal with Objects 7.0 в Delphi сохранен тип-объект `Object`, поддерживающий «старую» объектную модель. Поскольку все возможности этой модели доступны классам, я не буду рассматривать ее в этой книге, а «освободившийся» термин *объект* буду употреблять исключительно для обозначения конкретного экземпляра реализации класса.

Важным отличием классов от других типов является то, что объекты класса всегда распределяются в куче, поэтому объект-переменная фактически представляет собой лишь указатель на динамическую область памяти. Однако в отличие от других указателей, при ссылке на содержимое объекта *запрещается* использовать символ «^» за именем объекта:

```
type
  TMyClass = class(TObject)
    Field: Integer;
    ...
  end;
var
  MyClass: TMyClass;
begin
  ...
  MyClass^.Field := 0;      // Ошибка! Следует писать так:
  MyClass.Field := 0;
  ...
end;
```

Основные понятия

Классы — это особое «изобретение» программистов для упрощения разработки сложных программ и улучшения их качества. В основе классов лежат три фундаментальных принципа, которые называются *инкапсуляция*, *наследование* и *полиморфизм*.

Инкапсуляция

Класс представляет собой единство трех сущностей — полей, методов и свойств. Объединение этих сущностей в единое целое и называется инкапсуляцией. Инкапсуляция позволяет во многом изолировать класс от остальных частей программы, сделать его «самодостаточным» для решения конкретной задачи. В результате класс всегда несет в себе некоторую функциональность. Например, класс `TForm` содержит (инкапсулирует в себе) все необходимое для создания окна Windows-программы, класс `TMemo` представляет собой полнофункциональный многострочный текстовый редактор, класс `TTimer` обеспечивает работу программы с таймером и т. д.

Инкапсуляция является мощным средством обмена готовыми к работе программными заготовками. Библиотека классов Delphi — это, фактически, набор «кирпичиков», созданных программистами Borland для построения ваших программ.

Наследование

Любой класс может быть порожден от другого класса. Для этого при его объявлении указывается имя класса-родителя:

```
TChildClass = class(TParentClass)
```

Порожденный класс автоматически наследует поля, методы и свойства своего родителя и может добавлять новые. Таким образом, принцип наследования обеспечивает поэтапное создание сложных классов и разработку собственных библиотек классов.

Все классы Delphi порождены от единственного родителя — класса `TObject`. Этот класс не имеет полей и свойств, но включает в себя методы самого общего назначения, обеспечивающие весь жизненный цикл любых объектов — от создания до уничтожения. Программист не может создать класс, который не был бы дочерним классом `TObject`. Следующие два объявления идентичны:

```
TaClass = class(TObject)
```

```
TaClass = class
```

Принцип наследования приводит к созданию ветвящегося дерева классов, постепенно разрастающегося при перемещении от класса `TObject` к его потомкам. Каждый потомок дополняет возможности своего родителя новыми и передает их своим потомкам. Замечу, что дочерний класс не может удалить какую-либо сущность родительского класса. В отличие от C++, дочерний класс не может иметь два и более родительских классов, то есть в Delphi запрещено множественное наследование.

Для примера на рис. 8.1 показан небольшой фрагмент дерева классов Delphi. Класс `TPersistent` обогащает возможности своего родителя `TObject`: он «умеет» сохранять данные в файле и получать их из него, в результате это умеют делать и все его потомки. Класс `TComponent`, в свою очередь, умеет взаимодействовать со средой разработчика и передает это умение своим потомкам. Класс `TControl` не только способен работать с файлами и средой разработчика, он еще умеет создавать и обслуживать видимые на экране изображения, а его потомок `TWinControl` может создавать окна Windows-программ и т. д.

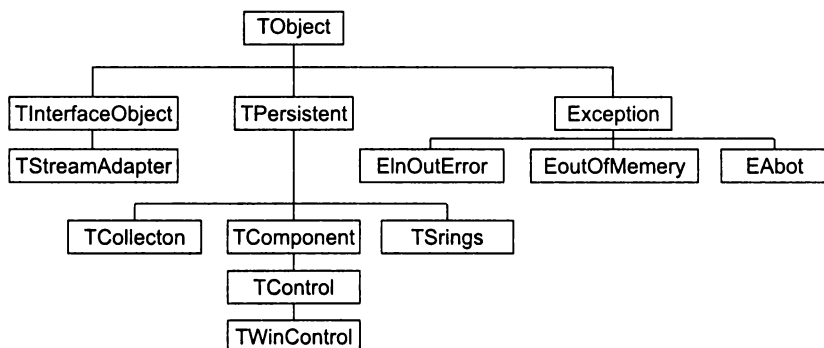


Рис. 8.1. Фрагмент дерева классов Delphi

Полиморфизм

Полиморфизм — это свойство классов решать схожие по смыслу проблемы разными способами. В рамках Delphi поведенческие свойства класса определяются набором входящих в него методов. Изменяя алгоритм того или иного метода в потомках класса, программист может придавать этим потомкам отсутствующие у родителя специфические свойства. Для изменения метода необходимо *перекрыть* его в потомке, то есть объявить в потомке одноименный метод и реализовать в нем нужные действия. В результате в объекте-родителе и объекте-потомке будут действовать два *одноименных* метода, имеющие разную алгоритмическую основу и, следовательно, придающие объектам разные свойства. Это и называется полиморфизмом объектов.

В Delphi полиморфизм достигается не только описанным выше механизмом наследования и перекрытия методов родителя, но и их *виртуализацией* (см. ниже), позволяющей родительским методам обращаться к методам своих потомков.

Составляющие класса

Поля

Полями называются инкапсулированные в классе данные. Поля могут быть любого типа, в том числе — классами, например:

```

type
  TMyClass = class

```

```

aIntField: Integer;
aStrField: String;
aObjField: TObject;
...

```

```
end;
```

Каждый объект получает уникальный набор полей, но общий для всех объектов данного класса набор методов и свойств. Фундаментальный принцип инкапсуляции требует обращаться к полям только с помощью методов и свойств класса. Однако в Delphi разрешается обращаться к полям и напрямую:

```
type
```

```

  TMyClass = class
    FIntField: Integer;
    FStrField: String;
    ...

```

```
end;
```

```
var
```

```
  aObject: TMyClass;
```

```
begin
```

```

  ...
  aObject.FIntField := 0;
  aObject.FStrField := 'Строка символов';
  ...

```

```
end;
```

Класс-потомок получает все поля всех своих предков и может дополнять их своими, но он не может переопределять их или удалять. Таким образом, чем ниже в дереве иерархии располагается класс, тем больше данных получают в свое распоряжение его объекты.

Методы

Инкапсулированные в классе процедуры и функции называются методами. Они объявляются так же, как и обычные подпрограммы:

```
type
```

```

  TMyClass = class
    Function MyFunc(aPar: Integer): Integer;
    Procedure MyProc;

```

```
end;
```

Доступ к методам класса, как и к его полям, возможен с помощью составных имен:

```
var
```

```
  aObject: TMyClass;
```

```
begin
```

```

  ...
  aObject.MyProc;
  ...

```

```
end;
```

ВНИМАНИЕ Любому методу класса доступна автоматически объявляемая переменная `Self`, которая трактуется как текущий экземпляр данного класса. Если, например, в классе объявлено целочисленное поле `IntData`, то два следующих присваивания в каком-либо методе класса идентичны:

```
IntData := 123;
Self.IntData := 123;
```

Как уже говорилось, методы класса могут перекрываться в потомках. Например:

```
type
  TParentClass = class
    Procedure DoWork;
  end;

  TChildClass = class (TParentClass)
    Procedure DoWork;
  end;
```

Потомки обоих классов могут выполнять сходную по названию процедуру `DoWork`, но, в общем случае, будут это делать по-разному. Такое перекрытие методов называется статическим, так как реализуется компилятором.

В Delphi гораздо чаще используется динамическое перекрытие методов на этапе прогона программы. Для реализации этого метод, перекрываемый в родительском классе, должен объявляться как динамический (с директивой **dynamic**) или виртуальный (**virtual**). Встретив такое объявление, компилятор создаст две таблицы — DMT (Dynamic Method Table) и VMT (Virtual Method Table) и поместит в них адреса точек входа соответственно динамических и виртуальных методов. При каждом обращении к перекрываемому методу компилятор вставляет код, позволяющий извлечь из той или иной таблицы адрес точки входа в подпрограмму. В классе-потомке перекрывающий метод объявляется с директивой **override** (перекрыть). Получив это указание, компилятор создаст код, который на этапе прогона программы поместит в родительскую таблицу точку входа метода класса-потомка, что позволит родителю выполнить нужное действие с помощью нового метода.

Пусть, например, родительский класс с помощью методов `Show` и `Hide` соответственно показывает что-то на экране или прячет изображение. Для создания изображения он использует метод `Draw` с логическим параметром:

```
type
  TVisualObject = class (TWinControl)
    Procedure Hide;
    Procedure Show;
    Procedure Draw(IsShow: Boolean); virtual;
  end;

  TVisualChildObject = class (TVisualObject)
```

```

Procedure Draw(IsShow: Boolean); override;
end;

```

Реализация методов Show и Hide очень проста:

```

Procedure TVisualObject.Show;
begin
  Draw(True);
end;

```

```

Procedure TVisualObject.Hide;
begin
  Draw(False);
end;

```

Методы Draw у родителя и потомка имеют разную реализацию и создают разные изображения. В результате родительские методы Show и Hide будут прятать или показывать те или иные изображения в зависимости от конкретной реализации метода Draw у любого из своих потомков. Динамическое связывание в полной мере реализует полиморфизм классов.

Разница между динамическими и виртуальными методами состоит в том, что таблица динамических методов (DMT) содержит адреса только тех методов, которые объявлены с директивой **dynamic** в данном классе, в то время как таблица VMT содержит адреса виртуальных методов не только данного класса, но и всех его родителей. Значительно большая по размеру таблица VMT обеспечивает более быстрый поиск, в то время как при обращении к динамическому методу программа сначала просматривает таблицу DMT у объекта, затем — у его родительского класса и так далее, пока не будет найдена нужная точка входа.

Динамически перекрываемые методы часто могут вообще ничего не делать. Такие методы называются абстрактными, они обязаны перекрываться в потомках. Программист может запретить вызов абстрактного метода, объявив его с директивой **abstract**. Например:

```

type
  TVisualObject = class(TWinControl)
    ...
    Procedure Draw(IsShow: Boolean); virtual; abstract;
  end;
  TVisualChildObject = class(TWinControl)
    ...
    Procedure Draw(IsShow: Boolean); override;
  end;
var
  aVisualObject: TVisualObject;
  aVisualChild: TVisualChildObject;
begin
  ...
  aVisualObject.Show; {Ошибка! Обращение к абстрактному
                       методу}

```



```
aVisualChild.Show; {Нормальное обращение. Метод Draw
                     у класса TVisualChildObject
                     перекрыт.}
```

```
...
end;
```

Обращение к неперекрытому абстрактному методу вызывает ошибку периода исполнения. Разумеется, в грамотно составленной программе абстрактные методы никогда не вызываются. Классы, содержащие абстрактные методы, называются абстрактными. Такие классы инкапсулируют общие свойства своих неабстрактных потомков, но объекты абстрактных классов никогда не создаются и не используются. Для эксплуатации абстрактных классов в библиотеку классов Delphi включаются классы-потомки, в которых перекрываются абстрактные методы родителя.

В состав любого класса входят два специальных метода — конструктор и деструктор. У класса TObject эти методы называются Create и Destroy, также они называются в подавляющем большинстве его потомков. Конструктор распределяет объект в динамической памяти (размещаются лишь поля объекта, его методы являются общими для всех объектов данного класса и в кучу не переносятся) и помещает адрес этой памяти в переменную Self, которая автоматически объявляется в классе. Деструктор удаляет объект из кучи. Обращение к конструктору должно предвещать любое обращение к полям и некоторым методам объекта. По своей форме конструкторы и деструкторы являются процедурами, но объявляются с помощью зарезервированных слов **Constructor** и **Destructor**:

```
type
  TMyClass = class
    IntField: Integer;
    Constructor Create(Value: Integer);
    Destructor Destroy;
  end;
```

Любые поля объекта, а также методы класса, оперирующие с его полями, могут вызываться только после создания объекта путем вызова конструктора, так как конструкторы распределяют объект в динамической памяти и делают действительным содержащийся в объекте указатель и автоматически объявляемую переменную Self.

```
var
  MyObject: TMyClass;
begin
  MyObject.IntField := 0; {Ошибка! Объект не создан
                          конструктором!}
  MyObject := TMyClass.Create; // Надо так: создаем объект
  MyObject.IntField := 0;      // и обращаемся к его полю
  ...
  MyObject.Free;               // Уничтожаем ненужный объект
end;
```

В базовом классе TObject определен метод Free, который сначала проверяет действительность адреса объекта и лишь затем вызывает деструктор Destroy.

Обращение к деструктору объекта будет ошибочным, если объект не создан конструктором, поэтому для уничтожения ненужного объекта следует вызывать метод `Free`, как это сделано в предыдущем примере.

Специально для уничтожения объектов в модуле `System` определена процедура `FreeAndNil`, которая не только уничтожает объект, но и помещает в его указатель (напомню, что им является идентификатор объекта) значение **NIL**:

```
FreeAndNil(MyObject)
```

Большинство конструкторов реализуют некоторые действия, необходимые для правильной работы объекта. Поэтому в конструкторе класса-потомка следует сначала вызвать конструктор своего родителя, а уже затем осуществлять дополнительные действия. Вызов любого метода родительского класса достигается с помощью зарезервированного слова **Inherited** (унаследованный):

```
Constructor TMyClass.Create(Value: Integer);
// Возможная реализация конструктора
begin
    Inherited Create; // Вызываем унаследованный конструктор
    IntField := Value; // Реализуем дополнительные действия
end;
```

Некоторые методы могут вызываться без создания и инициализации объекта. Такие методы называются методами класса, они объявляются с помощью зарезервированного слова **class**:

```
type
    TMyClass = class(TObject)
        class Function GetClassName: String;
    end;
var
    S: String;
begin
    S := TMyClass.GetClassName;
    ...
end;
```

Методы класса не должны обращаться к полям, так как в общем случае вызовы без создания объекта, а, следовательно, в момент вызова полей просто не существуют. Обычно они возвращают служебную информацию о классе — имя класса, имя его родительского класса, адрес метода и т. п.

Одноименные методы

В Delphi имеется возможность в рамках одного класса объявлять несколько *одноименных* методов. Описанный выше механизм перекрытия родительского метода одноименным методом потомка приводит к тому, что потомок «не видит» перекрытый родительский метод и может обращаться к нему лишь с помощью зарезервированного слова **Inherited**. В Delphi введено зарезервированное слово **override** (перезагрузить), с помощью которого становятся видны одноименные методы как родителя, так и потомка.

ВНИМАНИЕ Чтобы одноименные методы можно было отличить друг от друга, каждый из них должен иметь уникальный набор параметров. В ходе выполнения программы при обращении к одному из одноименных методов программа проверяет тип и количество фактических параметров обращения и выбирает нужный метод.

При обнаружении одноименного метода компилятор Delphi предупреждает о том, что у класса уже есть аналогичный метод с другими параметрами. Для подавления сообщений объявление одноименного метода можно сопровождать зарезервированным словом **reintroduce** (вновь ввести).

В следующем примере в классе `TForm1` используются целых 4 одноименных метода `Close`. Лишь один из них (унаследованный метод без параметра) выполняет свои основные функции — закрывает окно. Три другие отличаются набором параметров и выводят сообщение в заголовок окна.

Поместите на пустую форму четыре кнопки `TButton` и напишите такие обработчики их событий `OnClick`:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Close('Строка символов')
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    Close(123)
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
    Close(20, 300);
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
    Close
end;
```

Теперь в раздел **private** класса `TForm1` вставьте три объявления методов `Close`:

```
private
    { Private declarations }
    procedure Close(S: String); reintroduce; overload;
    procedure Close(I: Integer); reintroduce; overload;
    procedure Close(I,J: Integer); reintroduce; overload;
```

И, наконец, в разделе **implementation** поместите описания объявленных методов:

```
procedure TForm1.Close(S: String);
begin
```

```

Caption := S
end;

procedure TForm1.Close(I: Integer);
begin
    Caption := IntToStr(I)
end;

procedure TForm1.close(I,J: Integer);
begin
    Caption := IntToStr(i*j)
end;

```

Теперь после запуска программы три первые кнопки будут вызывать методы Close класса TForm1 и менять заголовок окна, в то время как кнопка Button4 обратится к методу Close родительского класса TForm и закроет окно.

Свойства

Свойства — это специальный механизм классов, регулирующий доступ к полям. Свойства объявляются с помощью зарезервированных слов **property**, **read** и **write** (слова **read** и **write** считаются зарезервированными только в контексте объявления свойства). Обычно свойство связано с некоторым полем и указывает те методы класса, которые должны использоваться при записи в это поле или при чтении из него. Например:

```

type
    TaClass = class
        IntField: Integer;
        Function GetField: Integer;
        Procedure SetField(Value: Integer);
        Property IntegerValue: Integer read GetField
        write SetField;
    end;

```

В контексте программы свойство ведет себя как обычное поле. Например, мы могли бы написать такие операторы:

```

var
    aClass: TaClass;
    Value: Integer;
begin
    aClass := TaClass.Create; { Обязательное обращение
                               к конструктору перед обращением к полю или свойству! }
    aClass.IntegerValue := 0;
    ...
    Value := aClass.IntegerValue;
    ...
    FreeAndNil(aClass);           // Удаление ненужного объекта
end;

```

Более того, возможен и такой оператор присваивания:

```
aClass.IntField := NewValue;
```

Разница между этим оператором и оператором, показанным ниже, заключается в том, что при обращении к свойству автоматически подключается метод `SetField`, в котором могут реализовываться специфичные действия:

```
aClass.IntegerValue := NewValue;
```

Вспомним использовавшийся нами в учебной программе оператор:

```
lbOutput.Caption := 'Строка';
```

Свойство `Caption` компонента `Label` вызывает метод `SetText`, который не только запоминает строку символов во внутренней переменной, но и осуществляет прорисовку метки с новым текстом.

Если нет необходимости в специальных действиях при чтении или записи свойства, вместо имени соответствующего метода можно указывать имя поля:

type

```
TaClass = class
  IntFiled: Integer;
  Procedure SetField(Value: Integer);
  Property IntegerValue:
    Integer read IntFiled write SetField;
end;
```

Если необходимо, чтобы свойство было доступно только для чтения или только для записи, следует опустить соответственно часть **write** или **read**. Вообще говоря, свойство может и не связываться с полем. Фактически оно описывает один или два метода, которые осуществляют некоторые действия над данными того же типа, что и свойство.

Объявление класса

Любой вновь создаваемый класс может содержать секции (разделы), определяемые зарезервированными словами **published** (опубликованные), **private** (закрытые), **protected** (защищенные), **public** (открытые) и **automated** (автоматизированные). Внутри каждой секции вначале определяются поля, а затем методы и свойства.

Секции определяют области видимости элементов описания класса. Секция **public** не накладывает ограничений на область видимости перечисляемых в ней полей, методов и свойств — их можно вызывать в любом другом модуле программы. Секция **published** также не ограничивает область видимости, однако в ней перечисляются свойства, которые должны быть доступны не только на этапе исполнения, но и на этапе конструирования программы (то есть в окне инспектора объектов). Секция **published** используется только при разработке нестандартных компонентов. Замечу, что среда Delphi помещает описания компонентов, вставленных в форму, в специальную секцию без названия, которая располагается сразу за заголовком класса и продолжается до первой объявленной секции. Эта секция — **published**. Программисту не следует помещать в нее собственные элементы описа-

ния класса или удалять из нее элементы, вставленные средой. Секция **private** сужает область видимости до минимума: закрытые элементы описания доступны только внутри методов данного класса и подпрограммах, находящихся в том же модуле, где описан класс. Элемент, объявленный в секции **private**, становится недоступным даже ближайшим потомкам класса, если они размещаются в других модулях. Секция **protected** доступна только методам самого класса, а также любым его потомкам, независимо от того, находятся ли они в том же модуле или нет. Наконец, секция **automated** используется только для объявления свойств и методов, которые будут добавлены к так называемому интерфейсу OLE-объектов автоматизации; область видимости членов этой секции не ограничена.

В Delphi разрешается сколько угодно раз объявлять любую секцию, причем порядок следования секций не имеет значения. Любая секция может быть пустой.

Следующий фрагмент кода поясняет различные области видимости:

```
Unit Unit1;
Interface
Uses Controls, Forms;
type
  TForm1 = class(TForm)
    Button1: TButton; // Эта секция обслуживается Delphi
                    // Ее элементы доступны всем
  private           // Эта секция доступна в модуле Unit1
    FIntField: Integer;
    Procedure SetValue(Value: Integer);
    Function GetValue: Integer;
  published        // Эта секция доступна в любом модуле
    Property IntField: read GetValue write SetValue;
  protected        // Эта секция доступна классам-потомкам
    Procedure Proc1;
  public            // Эта секция доступна в любом модуле
    Procedure Proc2;
  end;
var
  Form1: TForm1;
Implementation
Procedure TForm1.Proc1;
  Button1.Color := clBtnFace;           // Так можно
  FIntField := 0;                       // Так можно
  IntField := 0;                        // Так можно
  Proc1;                                // Так можно
  Proc2;                                // Так можно
end;
begin
  Form1.Button1.Color := clBtnFace; // Так можно
  Form1.FIntField := 0;             // Так можно
  Form1.IntField := 0;              // Так можно
  Form1.Proc1;                      // Так нельзя!
```

```

    Form1.Proc2;                                // Так можно
end.

Unit Unit2;
Interface
Uses Controls, Unit1;
type
    TForm2 = class(TForm1)
        Button2: TButton;
        Procedure Button2Click(Sender: TObject);
    end;
var
    Form2: TForm2;
Implementation
Procedure TForm2.Button2Click(Sender: TObject);
begin
    Button1.Color := clBtnFace;                // Так можно
    FIntField := 0;                            // Так нельзя!
    IntField := 0;                             // Так можно
    Proc1;                                     // Так можно
    Proc2;                                     // Так можно
end;
begin
    Form1.Button1.Color := clBtnFace; // Так можно
    Form1.FIntField := 0;             // Так нельзя!
    Form1.IntField := 0;              // Так можно
    Form1.Proc1;                      // Так нельзя!
    Form1.Proc2;                      // Так можно
end.

```

При объявлении класса-потомка разрешается перемещать элементы класса из одной области видимости в другую. Для предыдущего примера допустимо такое объявление:

```

type
    TForm2 = class(TForm1)
        ...
    Public
        procedure Proc1;
        ...
    end;

```

После этого в модуле Unit2 возможно следующее обращение:

```
Form2.Proc1;
```

После перемещения в секцию **private** элемент объявления становится невидим потомкам (если потомок, как это обычно бывает, объявляется в другом модуле) и, следовательно, его уже нельзя переместить в другую секцию.

Класс может объявляться только в интерфейсной области модуля или в самом начале области реализации. Нельзя определять классы в разделе описаний подпрограмм.

Приведение типов классов

Любой тип-класс относится к так называемым типам с информацией о типе во время исполнения (RunTime Type Information, RTTI). Экземпляры таких типов компилятор снабжает дополнительной информацией, позволяющей, в частности, на этапе прогона программы проверить принадлежность экземпляра нужному классу и осуществить его приведение. Для этого используются операторы **is** и **as**:

```
if (AObject is TMyClass) then  
    (AObject as TMyClass).IntData := 12345;
```


Глава 9

Варианты

Вариант — это тип `Variant`, разработанный специально для тех случаев, когда на этапе компиляции программист не может сказать, какого типа данные будут использоваться в выражении или в качестве параметров вызова подпрограмм. Переменная-вариант занимает в памяти дополнительные 2 байта, в которые помещается информация о действительном типе переменной. Эта информация позволяет компилятору создать код, который будет осуществлять необходимое преобразование типов на этапе прогона программы.

В переменную-вариант можно поместить:

- целое или вещественное число;
- логическое значение;
- строку;
- время и/или дату;
- OLE-объект;
- массив произвольной размерности и длины, содержащий элементы одного из перечисленных выше типов.

Варианты при условии корректности соответствующих преобразований могут участвовать в целочисленных, вещественных и логических выражениях, а также в выражениях с временем и датой. Например, если варианту `V` присвоена строка `'1.0'`, то выражение `1+V` будет правильным вещественным значением `2,0`. Однако если `V` содержит строку `'Текст'`, выражение `1+V` вызовет исключение `EVariantError`.

В Delphi определены перечисленные в табл. 9.1 константы, указывающие тип помещенных в вариант данных.

Таблица 9.1. Типы возможных значений варианта

Имя	Константа	Смысл
<code>varEmpty</code>	<code>\$0000</code>	Нет данных
<code>varNull</code>	<code>\$0001</code>	Вариант содержит <code>Null</code>
<code>varSmallInt</code>	<code>\$0002</code>	Целый тип <code>SmallInt</code>

Имя	Константа	Смысл
varInteger	\$0003	Целый тип Integer
varSingle	\$0004	Вещественный тип Single
varDouble	\$0005	Вещественный тип Double
varCurrency	\$0006	Вещественный тип Currency
varDate	\$0007	Тип дата-время
varOleStr	\$0008	OLE-строка в кодировке Unicode
varDispatch	\$0009	Указатель на OLE-объект
varError	\$000A	Код ошибки
varBoolean	\$000B	Тип WordBool
varVariant	\$000C	Тип Variant (только для вариантных массивов)
varUnknow	\$0011	Неизвестный OLE-объект
varByte	\$0100	Целый тип Byte
varString	\$0100	Строковый тип
varArray	\$2000	Вариантный массив
varByRef	\$4000	Указатель на данные

Структура вариантного типа описывается следующим образом:

```

TVData = packed record
  VType: Word;
  Reserved1, Reserved2, Reserved3: Word;
case Integer of
    varSmallInt: (VSmallInt: SmallInt);
    varInteger: (VInteger: Integer);
    varSingle: (VSingle: Single);
    varDouble: (VDouble: Double);
    varCurrency: (VCurrency: Currency);
    varDate: (VDate: Double);
    varOleStr: (VOleStr: PWideChar);
    varDispatch: (VDispatch: Pointer);
    varError: (VError: WordBool);
    varString: (VString: Pointer);
    varArray: (VArray: PVarArray);
    varByRef: (VPointer: Pointer);
end;

```

Как нетрудно убедиться, любая переменная вариантного типа представляет собой 16-байтную запись, содержащую 8-байтную вариантную часть, которая хранит либо собственно данные, либо их адрес (то есть указатель на динамически размещенные данные). В поле VType в момент создания варианта компилятор помещает признак отсутствия данных varEmpty. В работающей программе значение этого поля меняется в соответствии с текущим типом данных, размещенных в вариантной части. Замечу, что программа не может получить прямого доступа к по-

лям вариантной записи. Получить тип вариантных данных можно с помощью функции VarType (см. ниже), а изменить тип — путем присваивания варианту нового значения.

Преобразование вариантов к данным других типов

При участии вариантов в выражениях, а также при присваивании их значений переменным других типов тип размещенных в варианте данных преобразуется по правилам, указанным в табл. 9.2.

Таблица 9.2. Преобразование типов для вариантов

Приводится	Тип данных в варианте					
	<i>varEmpty</i>	<i>Целые</i>	<i>Вещественные</i>	<i>Дата-время</i>	<i>Строковые</i>	<i>Логические</i>
<i>К целым</i>	0	Преобразование в соответствующий тип	Округление до ближайшего целого	Округление до ближайшего целого	Преобразование в целый тип	0 для False, иначе -1 (255 для Byte)
<i>К вещественным</i>	0.0	Преобразование в соответствующий тип	Преобразование в соответствующий тип	Преобразование в соответствующий тип	Преобразование в вещественный тип	0 для False, -1 для True
<i>К типу дата-время</i>	30.12.1899 00:00:00	Преобразование в Double	Преобразование в Double	Без преобразования	Преобразование в дату	Преобразование в Double
<i>К строковым</i>	Пустая строка	Преобразование в символьный вид	Преобразование в символьный вид	Преобразование в символьный вид	Без преобразования	'0' для False, '-1' для True
<i>К логическим</i>	False	False для 0, иначе True	False для 0, иначе True	False для 0, иначе True	False для 'False' и для '0', иначе True	Без преобразования

Здесь:

- к целым отнесены типы varByte, varSmallInt, varInteger, varError;
- к вещественным отнесены типы — varSingle, varDouble, varCurrency;
- к строковым отнесены типы — varString, varOleStr.

Использование вариантов в выражениях

Подобно данным большинства других типов, варианты данные можно использовать в выражениях при условии корректности автоматического приведения типов. Если, например, варианты V1 и V2 содержат целые числа, то выражение V1+V2 будет целого типа, а выражение V1/V2 — вещественного. Однако что будет, если операнды-варианты содержат различные типы данных? Однозначно ответить на это трудно — все зависит от структуры выражения. Рассмотрим такой пример:

```

procedure TfmExample.bbRunClick(Sender: TObject);
var
    V1, V2, V3: Variant;
begin
    V1 := 10;    // Целый тип
    V2 := 20;    // Целый тип
    V3 := '30';  // Строковый тип
    lbOutput.Caption := V1+V2+V3
end;

```

Здесь сначала вычисляется выражение $V1+V2=30$, а затем к полученному целочисленному значению прибавляется строка, которая компилятором будет предварительно приведена к целому типу. В результате получим 60.

Изменим пример следующим образом:

```

procedure TfmExample.bbRunClick(Sender: TObject);
var
    V1, V2, V3: Variant;
begin
    V1 := '10'; // Строковый тип
    V2 := '20'; // Строковый тип
    V3 := 30;   // Целый тип
    lbOutput.Caption := V1+V2+V3
end;

```

Теперь сначала будет получена строка '1020', к которой следует прибавить целое число 30. Компилятор преобразует строку в целое число, в результате получим 1050. Если бы вычислялось следующее выражение, компилятор тут же привел бы каждый операнд к целому типу, и мы опять получили бы 60:

```
Integer(V1)+V2+V3
```

Об этих тонкостях применения вариантов не следует забывать: если вы используете выражения с операндами, везде, где это возможно, явно укажите тип результата. К сожалению, это противоречит самой природе варианта, и поэтому не всегда возможно.

Как уже отмечалось, вариант может иметь два особых значения: `varEmpty` и `varNull` (см. табл. 9.1). Первое он получает в момент создания. Оно означает, что вариант не имеет никакого значения. Второе означает, что значением варианта является *ничто* (`Null`). Это значение широко используется в приложениях для работы с базами данных. Для проверки, содержит ли вариант значение `varEmpty` или `varNull`, можно использовать две определенные в модуле `System` константы: `Unassigned` и `Null`. Если хотя бы один операнд выражения содержит значение `varNull`, все выражение приобретает такое же значение. Если какой-то операнд содержит значение `varEmpty`, вычисление выражения приведет к исключительной ситуации.

Подпрограммы для работы с вариантами

Для работы с вариантами можно использовать подпрограммы, перечисленные в табл. 9.3.

Таблица 9.3. Подпрограммы для работы с вариантами

Подпрограмма	Описание
function VarAsType(const V: Variant; VarType: Integer): Variant;	Преобразует данные варианта V к типу, определяемому параметром VarType
procedure VarCast(var Dest: Variant; const Source: Variant; VarType: Integer);	Преобразует данные варианта Source к типу, определяемому параметром VarType, и помещает результат в переменную Dest
procedure VarClear(var V: Variant);	Освобождает динамическую память, если она была связана с вариантом, и даст ему тип varEmpty
procedure VarCopy(var Dest: Variant; const Source: Variant);	Копирует параметр Source в вариант Dest
function VarFromDateTime(DateTime: TDateTime): Variant;	Возвращает вариант, содержащий данные DateTime типа дата-время
function VarIsEmpty(const V: Variant): Boolean;	Возвращает True, если вариант V не содержит данных
function VarIsNull(const V: Variant): Boolean;	Возвращает True, если вариант V содержит данные неопределенного типа (varNull)
function VarToDateTime(const V: Variant): TDateTime);	Преобразует данные варианта V к типу дата-время
function VarToStr(const V: Variant): String;	Преобразует данные варианта V к строке
function VarType(const V: Variant): Integer;	Возвращает тип хранящихся в варианте данных

Вариантные массивы

Значением варианта может быть массив данных, такие варианты называются вариантными массивами. (Не путайте с обычным или динамическим массивом, элементами которого являются варианты!) Значениями элементов вариантного массива могут быть любые допустимые для варианта значения, кроме строк varString. Значениями элементов вариантного массива могут быть и варианты, а это значит, что в таком массиве могут одновременно храниться данные разных типов (и в том числе строки). Например:

```

var
    V: Variant;
begin
    // Создаем одномерный вариантный массив с 5 элементами:
    V := VarArrayCreate([0, 4], varVariant);
    // Наполняем его:
    V[0] := 1;                                     // Целый тип

```

```

V[1] := 1234.5678           // Вещественный тип
V[2] := 'Hello world';      // Строковый тип
V[3] := True;               // Логический тип
// Пятым элементом исходного массива сделаем еще
// один массив:
V[4] := VarArrayOf([1, 10, 100, 1000]);
Caption := V[2];             // Hello world
lbOutput.Caption := IntToStr(V[4][2]); // 100
end;
```

Все действия с вариантными массивами осуществляются с помощью процедур и функций, перечисленных в табл. 9.4.

Таблица 9.4. Подпрограммы для работы с вариантными массивами

Подпрограмма	Описание
function VarArrayCreate(const Bounds: array of Integer; VarType: Integer): Variant;	Создает вариантный массив из элементов типа VarType с количеством и границами измерений, определяемыми параметром Bounds
function VarArrayDimCount(const A: Variant): Integer;	Возвращает количество измерений вариантного массива A или 0, если A — не массив
function VarArrayHighBound(const A: Variant; Dim: Integer): Integer;	Возвращает верхнюю границу индекса вариантного массива A по измерению Dim
function VarArrayLock(var A: Variant): Pointer;	Блокирует массив (предотвращает его возможные изменения размеров) и возвращает указатель на связанные с ним данные
function VarArrayLowBound(const A: Variant; Dim: Integer): Integer;	Возвращает нижнюю границу индекса вариантного массива A по измерению Dim
function VarArrayOf(const Values: array of Variant): Variant;	Создает одномерный вариантный массив по перечню значений, содержащихся в открытом массиве Values. Нижняя граница индексов вариантного массива в этом случае равна 0
procedure VarArrayRedim(var A: Variant; HighBound: Integer);	Изменяет верхнюю границу индекса вариантного массива A на величину HighBound. Вызов процедуры игнорируется, если массив был заблокирован функцией VarArrayLock
function VarArrayRef(const A: Variant): Variant;	Возвращает ссылку на вариантный массив. Используется при обращении к API-функциям
procedure VarArrayUnlock(var A: Variant)	Отменяет действие функции VarArrayLock

Пользовательские варианты

В Delphi можно создавать так называемые *пользовательские* варианты, которые, фактически, снимают ограничения на характер значений варианта.

Чтобы познакомиться со свойствами новых вариантов, воспользуемся одним из них — вариантом, способным хранить комплексные числа, преобразовывать их в другие типы и осуществлять над ними нужные действия. Как мы увидим дальше, создание пользовательского варианта может быть весьма трудоемким делом — все зависит от сложности хранимых в нем данных. Мы воспользуемся вариантом, созданным разработчиками Delphi и включенным в модуль VarCmplx — этот вариант способен хранить как стандартные для других вариантов значения (числа, строки, объекты OLE и т. д.), так и комплексные числа.

Создайте следующий обработчик bbRunClick (проект Source\Chap_09\Custom Variant\CustVar.dpr):

```
uses VarCmplx; // Эта ссылка обязательна!
procedure TfmExample.bbRunClick(Sender: TObject);
var
  V1, V2: Variant;
begin
  // Создаем два случайных комплексных числа:
  V1 := VarComplexCreate(Trunc(Random*1000)/100,
    Trunc(Random*1000)/100);
  V2 := VarComplexCreate(Trunc(Random*1000)/100,
    Trunc(Random*1000)/100);
  with mmOutput.Lines do
    begin // Пустая строка-разделитель
      Add('');
      Add('1-е число: '#9+V1);
      Add('2-е число: '#9+V2);
      Add('Сложение'#9+(V1+V2));
      Add('Вычитание'#9+(V1-V2));
      Add('Умножение'#9+(V1*V2));
      Add('Деление'#9#9+(V1/V2))
    end
  end;
```

Сложная конструкция `Trunc(Random*1000)/100` понадобилась только для того, чтобы реальные и мнимые части комплексных чисел содержали по три значащих цифры.

Вид экрана работающей программы показан на рис. 9.1.

Как видим, новый вариант легко справляется с поддержкой комплексных чисел: функция `VarComplexCreate` создает вариант, содержащий комплексное число, а дальнейшее поведение варианта стандартное (он поддерживает математические операции и преобразование к строковому типу). Однако эта легкость обманчива: исходный текст модуля `VarCmplx`, который, собственно, и придал варианту дополнительные свойства (по умолчанию располагается в файле

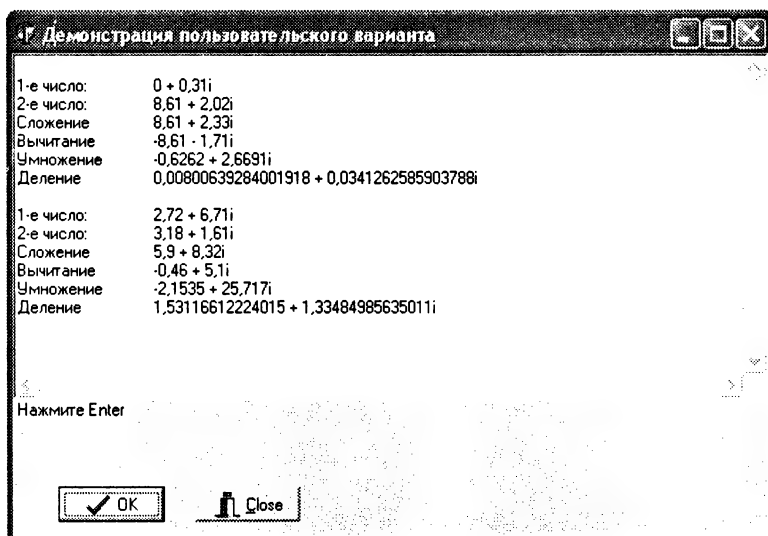


Рис. 9.1. Демонстрация комплексных вариантов

Source\Rtl\Common\VarCmplx.pas каталога размещения Delphi) содержит более 40 000 байт.

Методика создания пользовательских вариантов достаточно сложна и в этой книге не рассматривается.

Глава 10

Файлы

Под *файлом* понимается именованная область внешней памяти компьютера (жесткого диска, дискеты, компакт-диска и т. п.).

Любой файл имеет три характерные особенности. Во-первых, у него есть имя, что дает возможность программе работать одновременно с несколькими файлами. Во-вторых, он содержит компоненты одного типа. Типом компонентов может быть любой тип Delphi, кроме файлов. Иными словами, нельзя создать «файл файлов». В-третьих, длина вновь создаваемого файла никак не оговаривается при его объявлении и ограничивается только емкостью устройств внешней памяти.

Файловый тип можно задать одним из трех способов:

```
<имя> = File of <тип>;  
<имя> = TextFile;  
<имя> = File;
```

Здесь <имя> — имя файлового типа (правильный идентификатор); **File, of** — зарезервированные слова (файл, из); **TextFile** — имя стандартного типа текстовых файлов; <тип> — любой тип Delphi, кроме файлов.

Например:

```
type  
  Product = record  
    Name: String;  
    Code: Word;  
    Cost: Comp  
  end;  
  Text80 = File of String[80];  
var  
  F1: File of Char;  
  F2: TextFile;  
  F3: File;  
  F4: Text80;  
  F5: File of Product;
```

В зависимости от способа объявления можно выделить три вида файлов:

- типизированные файлы (задаются предложением **File of ...**);

- текстовые файлы (определяются типом `TextFile`);
- нетипизированные файлы (определяются типом `File`).

В представленном выше примере F1, F4 и F5 — типизированные файлы, F2 — текстовый файл, F3 — нетипизированный файл. Вид файла, вообще говоря, определяет способ хранения в нем информации. Однако в Delphi нет средств контроля вида ранее созданных файлов. При объявлении уже существующих файлов программист должен сам следить за соответствием вида объявления характеру хранящихся в файле данных.

Доступ к файлам

Файлы становятся доступны программе только после выполнения особой процедуры открытия файла. Эта процедура заключается в связывании ранее объявленной файловой переменной с именем существующего или вновь создаваемого файла, а также в указании направления обмена информацией: чтение из файла или запись в него.

Файловая переменная связывается с именем файла в результате обращения к стандартной процедуре `AssignFile`:

```
AssignFile (<ф.п.>, <имя файла>);
```

Здесь <ф.п.> — файловая переменная (правильный идентификатор, объявленный в программе как переменная файлового типа); <имя файла> — текстовое выражение, содержащее имя файла и, если это необходимо, маршрут доступа к нему.

Инициализировать файл означает указать для этого файла направление передачи данных. В Delphi можно открыть файл для чтения, для записи информации, а также для чтения и записи одновременно.

Для чтения файл инициализируется с помощью стандартной процедуры `Reset`:

```
Reset (<ф.п.>);
```

Здесь <ф.п.> — файловая переменная, связанная ранее процедурой `AssignFile` с уже существующим файлом.

При выполнении этой процедуры дисковый файл подготавливается к чтению информации. В результате специальная переменная-указатель, связанная с этим файлом, будет указывать на начало файла, то есть на компонент с порядковым номером 0.

В Delphi разрешается обращаться к типизированным файлам, открытым процедурой `Reset` (то есть для чтения информации), с помощью процедуры `Write` (то есть для записи информации). Такая возможность позволяет легко обновлять ранее созданные типизированные файлы и при необходимости расширять их. Для текстовых файлов, открытых процедурой `Reset`, нельзя использовать процедуру `Write` или `WriteLn`.

Следующая стандартная процедура иницирует запись информации в файл, связанный с файловой переменной <ф.п.>:

```
Rewrite (<ф.п.>);
```

СОВЕТ Если делается попытка инициировать чтение из несуществующего файла, возбуждается исключительная ситуация. Чтобы проверить, существует ли дисковый файл, можно использовать стандартную функцию `FileExists`, которая возвращает `True`, если указанный при обращении к этой функции файл существует, и `False` — если не существует. Например:

```
begin
  if FileExists(FileName) then
    ... // Файл существует
  else
    ... // Файл не существует
end;
```

Процедурой `Rewrite` нельзя инициировать запись информации в ранее существовавший дисковый файл: при выполнении этой процедуры старый файл (если он был) уничтожается и никаких сообщений об этом в программу не передается. Новый файл подготавливается к приему информации и его указатель принимает значение 0.

Следующая стандартная процедура иницирует запись в ранее существовавший текстовый файл для его расширения, при этом указатель файла устанавливается в его конец:

```
Append (<ô.ï.>)
```

Процедура `Append` применима только к текстовым файлам, то есть их файловой переменной должна иметь тип `TextFile` (см. выше). Процедурой `Append` нельзя инициировать запись в типизированный или нетипизированный файл. Если текстовый файл ранее уже был открыт с помощью процедуры `Reset` или `Rewrite`, использование процедуры `Append` приведет к закрытию этого файла и открытию его вновь, но уже для добавления записей.

Подпрограммы для работы с файлами

Ниже описываются процедуры и функции, которые часто используются с файлами любого вида (табл. 10.1). Некоторые другие подпрограммы для файлов перечислены в приложении Б. Специфика работы с типизированными, текстовыми и нетипизированными файлами рассматривается в следующих разделах.

Таблица 10.1. Подпрограммы для работы с файлами

Подпрограмма	Описание
procedure AssignFile (var F; FileName: String);	Связывает файловую переменную F с именем файла FileName
procedure CloseFile (var F);	Закрывает файл, однако связь файловой переменной F с именем файла, установленная ранее процедурой AssignFile, сохраняется. При создании нового или расширении старого файла процедура обеспечивает сохранение в файле всех новых записей и регистрацию

Подпрограмма	Описание
	файла в каталоге. Процедура CloseFile выполняется автоматически по отношению ко всем открытым файлам при нормальном завершении программы. Поскольку связь файла с файловой переменной сохраняется, файл можно повторно открыть без дополнительного использования процедуры AssignFile
function EOF(var F): Boolean;	Тестирует конец файла и возвращает True, если файловый указатель стоит в конце файла. При записи это означает, что очередной компонент будет добавлен в конец файла, при чтении — что файл исчерпан
procedure Erase(var F);	Уничтожает файл F. Перед выполнением процедуры необходимо закрыть файл. В ряде случаев вместо процедуры Erase удобнее использовать функцию DeleteFile, которая не требует предварительного связывания имени файла с файловой переменной.
function FileExists(const FileName: String): Boolean;	Возвращает True, если файл с именем (и, возможно, маршрутом доступа) FileName существует
function FindFirst (const Path: String ; Attr: Integer; var F: TSearchRec): Integer;	Возвращает атрибуты первого из файлов, зарегистрированных в указанном каталоге: Path — маршрут поиска и маска выбора файлов; Attr — атрибуты выбираемых файлов; F — переменная типа TSearchRec, в которой будет возвращено имя первого выбранного файла. При успешном поиске возвращает значение 0
procedure FindClose(var F: TSearchRec);	Освобождает память, выделенную для поиска файлов функциями FindFirst и FindNext
function FindNext(var F: TSearchRec): Integer;	Возвращает в переменной F имя следующего файла в каталоге. Переменная F должна предварительно инициализироваться обращением к функции FindFirst. При успешном поиске возвращает значение 0
procedure Flush(var F);	Очищает внутренний буфер файла и, таким образом, гарантирует сохранность всех последних изменений файла на диске
procedure GetDir(D: Byte; var S: String);	Возвращает имя текущего каталога (каталога по умолчанию): D — номер устройства (0 — устройство по умолчанию, 1 — диск A, 2 — диск B и т. д.); S — переменная типа String , в которой возвращается путь к текущему каталогу на указанном диске
procedure Mkdir(Dir: String);	Создает новый каталог на указанном диске: Dir — маршрут поиска каталога. Последним именем в маршруте, то есть именем вновь создаваемого каталога, не может быть имя уже существующего каталога

продолжение ➤

Таблица 10.1 (продолжение)

Подпрограмма	Описание
procedure Rename(var F; NewName: String);	Переименовывает файл F; NewName — строковое выражение, содержащее новое имя файла. Перед выполнением процедуры необходимо закрыть файл
procedure Reset(var F: File; RecSize: Word);	Открывает существующий файл. RecSize имеет смысл только для нетипизированных файлов и определяет размер блока данных
procedure Rewrite(var F: File [; RecSize: Word]);	Создает новый файл. RecSize имеет смысл только для нетипизированных файлов и определяет размер блока данных
procedure RmDir(Dir: String);	Удаляет каталог Dir. Удаляемый каталог должен быть пустым, то есть не содержать файлов или имен каталогов нижнего уровня

Подпрограммы FindFirst, FindNext и FindClose позволяют получить доступ к группе файлов, объединенных общими признаками. Эти признаки при обращении к функции FindFirst указываются маской выбора файлов и их атрибутами.

При формировании маски выбора файлов могут использоваться следующие символы-заменители:

- * — означает, что на месте этого символа может стоять сколько угодно (в том числе ноль) разрешенных символов имени или расширения файла;
- ? — означает, что на месте этого символа может стоять один из разрешенных символов.

Например:

- *.* — все файлы из каталога;
- c*.* — все файлы с именами, начинающимися на с (c1.pas, cc12345, c.dat и т. д.);
- a?.dat — имена файлов типа a0.dat, az.dat и т. д.

Маске выбора может предшествовать маршрут поиска файлов. Например, команда может выглядеть так

```
C:\Dir\SubDir\*.pas
```

Эта команда означает выбор всех файлов с расширением PAS из каталога SubDir, находящегося на диске C; каталог SubDir зарегистрирован в каталоге верхнего уровня Dir, который, в свою очередь, входит в корневой каталог. Если маршрут не указан, файлы ищутся в текущем каталоге.

Параметр Attr при обращении к FindFirst содержит двоичные разряды (биты), уточняющие, к каким именно файлам разрешен доступ. Вот как объявляются файловые атрибуты в модуле SysUtils:

```
const
    faReadOnly    = $01; // Только чтение
    faHidden      = $02; // Скрытый файл
    faSysFile     = $04; // Системный файл
```

```

faVolumeID = $08; // Идентификатор тома
faDirectory = $10; // Имя вложенного каталога
faArchive   = $20; // Архивный файл
faAnyFile   = $3F; // Любой файл

```

Комбинацией битов в этом байте можно указывать самые разные варианты, например \$06 — выбор всех скрытых и/или системных файлов.

Результат работы процедуры FindFirst возвращается в переменной типа TSearchRec. Этот тип определяется следующим образом:

```

type
  TSearchRec = record
    Time : Integer;
    Size : Integer;
    Attr : Integer;
    Name : TFileName;
    ExcludeAttr: Integer;
    FindHandle : THandle;
    FindDate   : Twin32FindDate;
  end;

```

Здесь Attr — атрибуты файла (см. выше); Time — время и дата создания или последнего обновления файла в системном формате; Size — длина файла в байтах; Name — имя и расширение файла; FindDate — переменная с дополнительной информацией о файле (время создания, время последнего доступа).

Результат обращения к процедуре FindFist возвращается в значении типа Integer, которое равно 0, если нет ошибок.

Следующая простая программа иллюстрирует способ использования функций FindFirst и FindNext. Программа выводит в окно многострочного текстового поля mmOutput список всех файлов, маска выбора которых (и, возможно, маршрут поиска) указана в поле edInput (см. проект Source\Chap_10\FindFile\FindFile.dpr):

```

procedure TfmExample.bbRunClick(Sender: TObject);
var
  Mask: String;
  SR: TSearchRec;
begin
  Mask := edInput.Text;
  if Mask = '' then
    Mask := '*.*';
  mmOutput.Lines.Clear;
  if FindFirst(Mask, faAnyFile, SR)=0 then
    repeat
      mmOutput.Lines.Add(SR.Name);
    until FindNext(SR) <> 0;
  FindClose(SR);
end;

```

Окно работающей программы представлено на рис. 10.1.

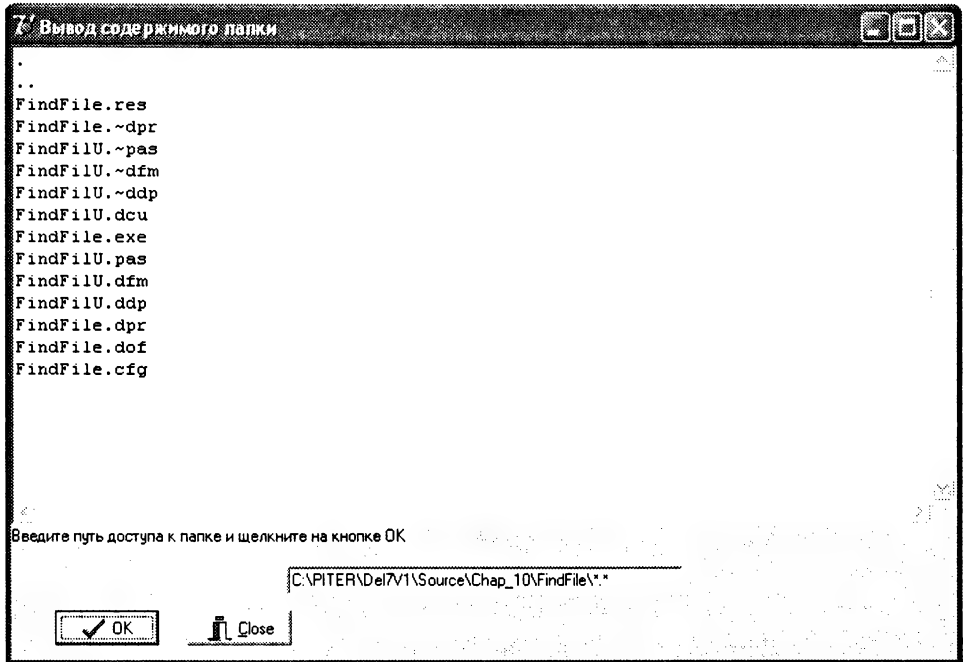


Рис. 10.1. Отображение списка файлов

Любое обращение к файлу в Delphi осуществляется через некоторый буфер, что необходимо для согласования внутреннего представления файлового компонента (записи) с принятым в операционной системе форматом хранения данных на диске. В ходе выполнения процедуры `Flush` все новые записи будут действительно записаны на диск. Процедура игнорируется, если файл был инициализирован для чтения процедурой `Reset`.

В Delphi для защиты программы от краха при выполнении потенциально опасных фрагментов широко используется механизм обработки исключительных ситуаций (см. главу 13). В следующем фрагменте показано, как можно использовать этот механизм при работе с файлами. Предположим, что требуется отредактировать файл, имя которого содержит переменная `Name`. Перед редактированием необходимо убедиться, что нужный файл имеется на диске, и создать его страховочную копию с расширением `BAK`. Если одноименный файл (то есть с таким же именем и расширением `BAK`) уже существует, его надо уничтожить.

```
var
  Fi : TextFile;      // Исходный файл
  Fo : TextFile;      // Отредактированный файл
  Name : String;      // Имя исходного файла
  Name_bak : String;  // Имя страховочного файла
const
  BAK = '.bak';
begin
  ...
```

```

// Получаем в name_bak имя файла с расширением .BAK:
Name_bak := copy(Name, 1, pos('.', Name)) + BAK;
// Проверяем существование исходного файла:
AssignFile(Fi, Name);
try
    Reset(Fi);
except
    Halt; // Завершаем программу: файла не существует
end;
CloseFile(Fi);
// Проверяем существование BAK-файла:
AssignFile(Fo, Name_bak);
try
    Reset(Fo);
    // BAK-файл существует:
    CloseFile(Fo); // Закрываем его
    Erase(Fo)      // и уничтожаем
finally
    // Файл не существует: ничего не делаем
end;
// Проверки закончены, подготовка к работе:
Rename(Fi, Name_bak);
Reset(Fi);
AssignFile(Fo, Name);
Rewrite(Fo);
...
end.

```

Обратите внимание: проверка на существование BAK-файла в данном примере необходима, так как следующее обращение вызовет ошибку в случае, если такой файл существует:

```
Rename(Fi, Name_bak);
```

Текстовые файлы

Текстовые файлы связываются с файловыми переменными, принадлежащими стандартному типу `TextFile`. Текстовые файлы предназначены для хранения текстовой информации. Именно в такого типа файлах хранятся, например, исходные тексты программ. Компоненты (записи) текстового файла могут иметь переменную длину, что существенно влияет на характер работы с ними.

Текстовый файл трактуется в Delphi как совокупность строк переменной длины. Доступ к каждой строке возможен лишь последовательно, начиная с первой. При создании текстового файла в конце каждой строки ставится специальный признак `EOLN` (End Of LiNe — конец строки), а в конце всего файла — признак `EOF` (End Of File — конец файла). Эти признаки можно протестировать одноименными логическими функциями (см. ниже). При формировании текстовых файлов используются следующие системные соглашения:

- ☒ EOLN — последовательность кодов #13 (CR) и #10 (LF);
- ☒ EOF — код #26.

Для доступа к записям применяются процедуры `Read`, `ReadLn`, `Write`, `WriteLn`. Они отличаются возможностью обращения к ним с переменным количеством фактических параметров, в качестве которых могут использоваться символы, строки и числа. Первым параметром в любой из перечисленных процедур должна стоять файловая переменная. Обращение осуществляется к дисковому файлу, связанному с переменной процедурой `AssignFile`.

В табл. 10.2 указаны подпрограммы для работы с текстовыми файлами.

Таблица 10.2. Подпрограммы для работы с текстовыми файлами

Подпрограмма	Описание
function Eoln(var F: TextFile): Boolean;	Проверяет признак конца строки и возвращает True, если конец строки достигнут
procedure Read(var F: TextFile; V1 [, V2, ..., Vn]);	Читает из текстового файла последовательность символьных представлений переменных Vi типа Char, String , а также любого целого или вещественного типа, игнорируя признаки EOLN
procedure ReadLn (var F: TextFile; [V1 [, V2, ..., Vn]]);	Читает из текстового файла последовательность символьных представлений переменных Vi типа Char, String , а также любого целого или вещественного типа с учетом границ строк
function SeekEof(var F: Text): Boolean;	Пропускает все пробелы, знаки табуляции и признаки конца строки EOLN до признака конца файла EOF или до первого значащего символа и возвращает True, если признак EOF обнаружен
function SeekEoln (var F: TextFile): Boolean;	Пропускает все пробелы и знаки табуляции до признака конца строки EOLN или до первого значащего символа и возвращает True, если признак обнаружен
procedure Write(var F: Text; P1 [, P2, ..., Pn]);	Записывает символьные представления параметров Pi в текстовый файл
procedure WriteLn (var F: Text; [P1 [, P2, ..., Pn]]);	Записывает символьные представления параметров Pi и признак конца строки EOLN в текстовый файл

Процедура `Read` предназначена для последовательного чтения из текстового файла символьных представлений переменных Vi. При чтении переменных типа Char выполняется чтение одного символа и присваивание считанного значения переменной. Если перед выполнением чтения указатель файла достиг конца очередной строки, то результатом чтения будет символ CR (код #13), а если достигнут конец файла, то — символ EOF (код #26). Процедуру `Read` не рекомендуется использовать для ввода переменных типа **String**, так как она не способна «перепрыгнуть» через разделитель строк EOLN и читает только первую строку текстового файла. Для ввода последовательности строк нужно использовать процедуру

ReadLn (см. ниже). Следующая программа «зависнет», так как вторая строка файла никогда не будет прочитана:

```
procedure TfmExample.bbRunClick(Sender: TObject);
var
  F: TextFile;
  S: String;
begin
  AssignFile(F, 'example.pas');
  Reset(F);
  while not EOF(F) do
    begin
      Read(F,S); // Ошибка! Бесконечный цикл!
      mmOutput.Lines.Add(S)
    end;
  CloseFile(F)
end;
```

При вводе численных переменных процедура Read вначале выделяет подстроку во входном потоке по следующему правилу: все ведущие пробелы, символы табуляции и признаки конца строк EOLN пропускаются; после выделения первого значащего символа, наоборот, любой из перечисленных символов или символ EOF служат признаком конца подстроки. Выделенная таким образом подстрока затем рассматривается как символьное представление числовой константы соответствующего типа и преобразуется во внутреннее представление, а полученное значение присваивается переменной. Если в подстроке был нарушен требуемый формат представления числовой константы, возникает исключительная ситуация. Если при пропуске ведущих пробелов встретился символ EOF, переменная получает значение 0. В Delphi не предусмотрен ввод шестнадцатеричных констант.

Процедура Read прекрасно приспособлена к вводу чисел. При обращении к ней за вводом очередного целого или вещественного числа процедура «перескакивает» признаки конца строк, то есть фактически весь файл рассматривается ею как одна длинная строка, содержащая текстовые представления чисел. В сочетании с проверкой конца файла функцией EOF процедура Read позволяет организовать простой ввод массивов данных, например, так:

```
const
  N = 1000; // Максимальное количество вводимых символов
var
  F : TextFile;
  M : array [1..N] of Real;
  i : Integer;
begin
  AssignFile(F, 'prog.dat');
  Reset(F);
  i := 1;
  while not EOF(f) and (i <= N) do
    begin
```

```

    Read(F, M[i]);
    inc(i)
end;
CloseFile(F);
...
end.

```

Процедура `ReadLn` идентична процедуре `Read`, за исключением того, что после считывания последней переменной оставшаяся часть строки до признака `EOLN` пропускается, поэтому следующее обращение к `ReadLn` начинается с первого символа новой строки. Кроме того, эту процедуру можно вызвать без параметров `Vi`, что приведет к пропуску всех символов текущей строки вплоть до `EOLN`. Если в обработчике `bbRunClick` (см. пример на предыдущей странице) заменить `Read` на `ReadLn`, программа выведет в окно компонента `mmOutput` все строки из текстового файла `EXAMPLE.PAS`.

Процедура `Write` обеспечивает вывод в текстовый файл группы переменных. Любой элемент списка вывода может иметь следующую форму:

```
OutExpr [ : MinWidth [ : DecPlaces ] ]
```

Здесь `OutExpr` — выводимое выражение; `MinWidth`, `DecPlaces` — выражения типа `Integer` (квадратные скобки означают возможность отсутствия заключенных в них параметров). Параметр `MinWidth`, если он присутствует, определяет минимальную ширину поля, в которое будет записываться символьное представление значения `OutExpr`. Если символьное представление имеет меньшую длину, чем `MinWidth`, оно будет дополнено слева пробелами, если большую длину, то `MinWidth` игнорируется и в файл помещается необходимое количество символов. Параметр `DecPlaces` задает количество десятичных знаков в дробной части вещественного числа. Он может использоваться только совместно с `MinWidth` и только по отношению к выводимому выражению одного из вещественных типов.

Если ширина поля вывода не указана, соответствующий параметр выводится вслед за предыдущим без какого-либо их разделения. Символы и строки передаются выходному файлу без изменений, но снабжаются ведущими пробелами, если задана ширина поля вывода и эта ширина больше требуемой для вывода.

При выводе логических выражений в зависимости от их значения выводятся слова `True` или `False`. (Ввод логических констант процедурами `Read` или `ReadLn` не предусмотрен.)

Вещественные числа выводятся в экспоненциальном формате, если не указан параметр `DecPlaces`, в противном случае выбирается формат представления числа с фиксированной точкой. Если подпараметр `MinWidth` опущен, принимается его значение по умолчанию (23). Если `MinWidth` меньше 10, считается, что он равен 10. Если подпараметр `DecPlaces` равен нулю, ни дробная часть числа, ни десятичная точка не выводятся. При отрицательном значении `DecPlaces` этот параметр игнорируется, и число выводится в экспоненциальном формате с учетом `MinWidth`. Если значение `DecPlaces` больше 18, принимается значение 18. Следует учесть, что при указании подпараметра `DecPlaces` вещественное число всегда будет выводиться в формате с фиксированной точкой и требуемым количеством знаков в дробной части, даже если значение подпараметра `MinWidth` окажется

недостаточным для размещения целой части: в этом случае значение `MinWidth` автоматически увеличивается.

Процедура `WriteLn` полностью идентична процедуре `Write`, за исключением того, что выводимая последовательность символов автоматически завершается признаком `EOLN` (свое название процедура получила от *Write Line* — писать строку). При вызове `WriteLn` можно опускать параметры `Vi` — в этом случае в файл передается пустая строка.

Типизированные файлы

Длина любого компонента типизированного файла строго постоянна, что дает возможность организовать прямой доступ к каждому из них (то есть доступ к компоненту по его порядковому номеру).

Перед первым обращением к процедурам ввода-вывода указатель файла стоит в его начале и указывает на первый компонент с номером 0. После каждого чтения или записи указатель сдвигается к следующему компоненту файла. Переменные в списках ввода-вывода должны иметь тот же тип, что и компоненты файла. Если этих переменных в списке несколько, указатель будет смещаться после каждой операции обмена данными между переменными и дисковым файлом.

В табл. 10.3 указаны подпрограммы для работы с типизированными файлами.

Таблица 10.3. Подпрограммы для работы с типизированными файлами

Подпрограмма	Описание
function FilePos (var F): LongInt;	Возвращает текущую позицию в файле, то есть номер компонента, который будет обрабатываться следующей операцией ввода-вывода
function FileSize (var F): LongInt;	Возвращает количество компонентов файла. Чтобы переместить указатель в конец типизированного файла, можно написать <code>seek (FileVar, FileSize (FileVar))</code> ;
procedure Seek (var F; N: LongInt);	Смещает указатель файла F к требуемому компоненту: N — номер компонента файла (первый компонент файла имеет номер 0)
procedure Read (var F, V1, ..., Vn);	Читает данные из типизированного файла F: Vi — переменные такого же типа, что и компоненты файла
procedure Write (var F, P1, ..., Pn);	Записывает данные в типизированный файл F: Pi — выражения такого же типа, что и компоненты файла

Нетипизированные файлы

Нетипизированные файлы объявляются как файловые переменные типа **File** и отличаются тем, что для них не указан тип компонентов. Отсутствие типа делает эти файлы, с одной стороны, совместимыми с любыми другими файлами, а с другой — позволять организовать высокоскоростной обмен данными между диском и памятью.

При инициализации нетипизированного файла процедурами `Reset` или `Rewrite` можно указать длину записи нетипизированного файла в байтах. Например, так:

```
var
  F: File;
begin
  ...
  AssignFile (F, 'myfile.dat');
  Reset (f, 512);
  ...
end.
```

Длина записи нетипизированного файла указывается вторым параметром при обращении к процедурам `Reset` или `Rewrite`, в качестве которого может использоваться выражение типа `LongInt`. Если длина записи не указана, она принимается равной 128 байт.

Delphi не накладывает каких-либо ограничений на длину записи нетипизированного файла, за исключением требования положительности и ограничения максимальной длины значением 2 Гбайт (для Delphi 1 длина записи ограничивается значением 65 535 байт). Для обеспечения максимальной скорости обмена данными рекомендуется задавать длину, которая была бы кратна длине физического сектора дискового носителя информации (512 байт). Однако операции обмена данными с дисковыми устройствами в среде Windows кэшируются, то есть осуществляются через промежуточный буфер памяти, поэтому обычно задают `RecSize = 1`, что позволяет обмениваться с файлом блоками любой длины, начиная с одного байта.

При работе с нетипизированными файлами могут применяться все процедуры и функции, доступные типизированным файлам, за исключением `Read` и `Write`, которые заменяются соответственно высокоскоростными процедурами `BlockRead` и `BlockWrite`:

```
procedure BlockRead(var F: File; var Buf; Count: Integer [;
var AmtTransferred: Integer]);
procedure BlockWrite(var F: File; var Buf; Count: Integer
[; var AmtTransferred: Integer]);
```

Здесь `Buf` — буфер, то есть имя переменной, которая будет участвовать в обмене данными с дисками; `Count` — количество записей, которые должны быть прочитаны или записаны за одно обращение к диску; `AmtTransferred` — необязательный параметр, содержащий при выходе из процедуры количество фактически обработанных записей.

За одно обращение к процедурам может быть передано до `Count×RecSize` байтов, где `RecSize` — длина записи нетипизированного файла. Передача идет, начиная с первого байта переменной `Buf`. Программист должен позаботиться о том, чтобы длина внутреннего представления переменной `Buf` была достаточной для размещения всех `Count×RecSize` байтов при чтении информации с диска. Если при чтении указана переменная недостаточной длины или если в процессе записи на диск не окажется нужного свободного пространства, возникнет ошибка ввода-

вывода, которую можно заблокировать, указав необязательный параметр `Amt-Transferred`.

После завершения процедуры указатель смещается на `Count` записей. Процедурями `Seek`, `FilePos` и `FileSize` можно обеспечить доступ к любой записи нетипизированного файла.

Отображение файлов в память

Для работы с файлом динамической подкачки страниц виртуальной памяти в 32-разрядных версиях Windows используется механизм отображения файлов в адресное пространство программы. Соответствующие функции API доступны любой программе и могут применяться к любому файлу (кстати, таким способом загружаются в адресное пространство процесса исполняемые файлы). В результате отображения программа может работать с файловыми данными как с данными, размещенными в динамической памяти. Такая возможность не только в большинстве случаев повышает скорость работы с данными, но и предоставляет программисту уникальные средства обработки сразу всех записей файла. Например, он может единственным оператором проверить, входит ли заданный образец поиска в любую строку текстового файла.

Отображение файла осуществляется в три приема. Вначале файл создается обращением к следующей функции:

```
function FileCreate (FileName: String): Integer;
```

Если файл уже существует, он открывается с помощью другой функции:

```
function FileOpen (const FileName: String; Mode: LongWord):  
Integer;
```

В обеих функциях `FileName` — имя файла, возможно, с маршрутом доступа. Параметр `Mode` определяет режим доступа к файлу и может принимать одно из следующих значений:

- ❏ `fmOpenRead` — только чтение;
- ❏ `fmOpenWrite` — только запись;
- ❏ `fmOpenReadWrite` — чтение и запись.

С помощью операции **or** эти константы можно комбинировать с одной из следующих, регулирующих совместный доступ к файлу нескольких программ:

- ❏ `fmShareExclusive` — совместный доступ запрещен;
- ❏ `fmShareDenyWrite` — другим программам запрещается запись;
- ❏ `fmShareDenyRead` — другим программам запрещается чтение;
- ❏ `fmShareDenyNone` — совместный доступ неограничен.

Обе функции возвращают дескриптор созданного (открытого) файла или 0, если операция оказалась неуспешной.

На втором этапе создается объект отображения в память. Для этого используется такая функция:

```
function CreateFileMapping (hFile: THandle;  
lpFileMappingAttributes: PSecurityAttributes;
```

```
flProtect, dwMaximumSizeHigh, dwMaximumSizeLow: DWord;
lpName: PChar): THandle;
```

Параметры здесь следующие:

- `hFile` — дескриптор файла;
- `lpFileMappingAttributes` — указатель на структуру, в которой определяется, может ли создаваемый объект порождать дочерние объекты (обычно не может, и параметр равен **NIL**);
- `flProtect` — определяет тип защиты, применяемый к окну отображения файла; допустимые значения параметра (см. также примечание ниже):
 - ☐ `PAGE_READONLY` — файл можно только читать (файл должен быть создан или открыт в режиме `fmOpenRead`);
 - ☐ `PAGE_READWRITE` — файл можно читать и записывать в него новые данные (файл открывается в режиме `fmOpenReadWrite`);
 - ☐ `PAGE_WRITECOPY` — файл открыт для записи и чтения, однако обновленные данные сохраняются в отдельной защищенной области памяти (отображенные файлы могут разделяться программами, в этом режиме каждая программа сохраняет изменения в отдельной области памяти или участке файла подкачки; файл открывается в режиме `fmOpenReadWrite` или `fmOpenWrite`); этот тип защиты нельзя использовать в Windows 95/98;
- `dwMaximumSizeHigh` и `dwMaximumSizeLow` — соответственно старшие и младшие 32 разряда размера файла; если вы будете отображать файлы длиной до 4 Гбайт, поместите в `dwMaximumSizeHigh` 0, а в `dwMaximumSizeLow` — длину файла; если оба параметра равны 0, размер окна отображения равен размеру файла;
- `lpName` — имя объекта отображения или **NIL**.

ПРИМЕЧАНИЕ

С помощью операции **or** к параметру `flProtect` можно присоединить такие атрибуты: `SEC_COMMIT` — выделяет для отображения физическую память или участок файла подкачки; `SEC_IMAGE` — информация об атрибутах отображения берется из образа файла; `SEC_NO-CACHE` — отображаемые данные не кэшируются и записываются непосредственно на диск; `SEC_RESERVE` — резервируются страницы раздела без выделения физической памяти.

Функция возвращает дескриптор объекта отображения или 0, если обращение было неудачным.

Наконец, на третьем этапе создается окно просмотра, то есть собственно отображение данных в адресное пространство программы:

```
function MapViewOfFile(hFileMappingObject: THandle;
dwDesiresAccess: DWord; dwFileOffsetHigh, dwFileOffsetLow,
dwNumberOfBytesToMap: DWord): Pointer;
```

Параметры здесь следующие:

- `hFileMappingObject` — дескриптор объекта отображения;

- ❖ `dwDesiresAccess` — определяет способ доступа к данным и может иметь одно из следующих значений:
 - ❑ `FILE_MAP_WRITE` — разрешает чтение и запись, при этом в функции `CreateFileMapping` должен использоваться атрибут `PAGE_READWRITE`;
 - ❑ `FILE_MAP_READ` — разрешает только чтение, в функции `CreateFileMapping` должен использоваться атрибут `PAGE_READONLY` или `PAGE_READWRITE`;
 - ❑ `FILE_MAP_ALL_ACCESS` — то же, что и `FILE_MAP_WRITE`;
 - ❑ `FILE_MAP_COPY` — данные доступны для записи и чтения, однако обновленные данные сохраняются в отдельной защищенной области памяти; в функции `CreateFileMapping` должен использоваться атрибут `PAGE_WRITECOPY`;
- ❖ `dwFileOffsetHigh` и `dwFileOffsetLow` — определяют соответственно старшие и младшие разряды смещения от начала файла, начиная с которого осуществляется отображение;
- ❖ `dwNumberOfBytesToMap` — определяет длину окна отображения (0 — длина равна длине файла).

Функция возвращает указатель на первый байт отображенных данных или **NULL**, если обращение к функции оказалось неуспешным.

После использования отображенных данных ресурсы окна отображения нужно освободить следующей функцией:

```
function UnMapViewOfFile(lpBaseAddress: Pointer): BOOL;
```

Единственный параметр обращения к этой функции должен содержать адрес первого отображенного байта, то есть адрес, возвращаемый функцией `MapViewOfFile`. Закрытие объекта отображения и самого файла осуществляется обращением к другой функции:

```
function CloseHandle(hObject: THandle)
```

В листинге 10.1 (проект `Chap_10\File Mapping\FileInMemory.dpr`) приводится текст модуля, который создает окно, показанное на рис. 10.2.

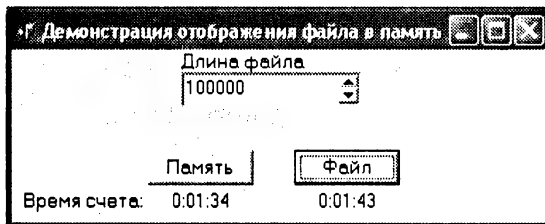


Рис. 10.2. Демонстрация отображения файла

В проекте создается дисковый файл, состоящий из 100 000 случайных вещественных чисел (длину файла можно выбрать другой, если изменить значение в поле со счетчиком `Длина файла`). Файл с именем `test.dat` создается отображением файла в память (кнопка `Память`) и традиционным способом (кнопка `Файл`). В обоих слу-

чаях отображается время счета. Чем больше частота и память, тем больше будет разница во времени.

Листинг 10.1. Иллюстрация отображения файла в память¹

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ComCtrls, Spin;

type
  TForm1 = class (TForm)
    btMem: TButton;
    btFile: TButton;
    se: TSpinEdit;
    Label1: TLabel;
    pb: TProgressBar;
    Label2: TLabel;
    lbMem: TLabel;
    lbFile: TLabel;
    procedure btMemClick(Sender: TObject);
    procedure btFileClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.btMemClick(Sender: TObject);
  // Создание файла методом его отображения
type
  PReal = ^Real;
var
  HFile, HMap: THandle;

```

¹ Все представленные в книге тексты программ можно загрузить с сайта издательства по адресу <http://www.piter.com/download> — Примеч. ред.

```

AdrBase, AdrReal: PReal;
k: Integer;
FSize: Cardinal;
BegTime: TDateTime;
begin
    BegTime := Time;    // Засаекаем время пуска
    // Готовим ProgressBar:
    pb.Max := se.Value;
    pb.Position := 0;
    pb.Show;
    FSize := se.Value * SizeOf(Real); // Длина файла
    HFile := FileCreate('test.dat'); // Создаем файл
    if HFile = 0 then // Ошибка: возбуждаем исключение
        raise Exception.Create('Ошибка создания файла');
    try
        // Отображаем файл в память
        HMap := CreateFileMapping(HFile, NIL, PAGE_READWRITE,
            0, FSize, NIL);
        if HMap = 0 then // Ошибка: возбуждаем исключение
            raise Exception.Create('Ошибка отображения файла');
        try
            // Создаем окно просмотра:
            AdrBase := MapViewOfFile(HMap, FILE_MAP_WRITE, 0, 0,
                FSize);
            if AdrBase = NIL then // Ошибка: возбуждаем
                // исключение
                raise Exception.Create('Невозможно просмотреть файл');
            // Сохраняем начальный адрес для правильной
            // ликвидации окна просмотра:
            AdrReal := AdrBase;
            for k := 1 to se.Value do
                begin
                    AdrReal^ := Random; // Помещаем в файл новое число
                    // Перед наращиванием текущего адреса необходимо
                    // привести его к типу Integer или Cardinal:
                    AdrReal := Pointer(Integer(AdrReal) +
                        SizeOf(Real));
                    lbMem.Caption := IntToStr(k);
                    pb.Position := k;
                    Application.ProcessMessages;
                end;
            // Освобождаем окно просмотра:
            UnmapViewOfFile(AdrBase)
        finally
            // Освобождаем отображение
            CloseHandle(HMap)
        end
    end
end

```

Листинг 10.1 (продолжение)

```

    end
  finally
    // Закрываем файл
    CloseHandle(HFile)
  end;
  // Сообщаем время счета
  pb.Hide;
  lbMem.Caption := TimeToStr(Time-BegTime)
end;

procedure TForm1.btFileClick(Sender: TObject);
// Создание файла обычным методом
var
  F: File of Real;
  k: Integer;
  BegTime: TDateTime;
  R: Real;    // Буферная переменная для обращения к Write
begin
  BegTime := Time; // Засекаем начальное время счета
  // Готовим ProgressBar:
  pb.Max := se.Value;
  pb.Position := 0;
  pb.Show;
  // Создаем файл:
  AssignFile(F, 'test.dat');
  Rewrite(F);
  for k := 1 to se.Value do
  begin
    R := Random;    // Параметрами обращения к Write
    Write(F, R);    // могут быть только переменные
    lbFile.Caption := IntToStr(k);
    pb.Position := k;
    Application.ProcessMessages;
  end;
  CloseFile(F);
  pb.Hide;
  lbFile.Caption := TimeToStr(Time-BegTime)
end;

end.

```

Следует заметить, что в программе используются защищенные блоки **try ... finally ... end**, о которых рассказывается в главе 13.

Объектная модель работы с файлами

В Delphi используется абстрактный класс `TStream` (поток данных), который является основой для работы с файлами как с объектами. В случае объектов совершенно неважно, что именно является носителем информации — дисковый ли файл, ленточный носитель или оперативная память. В специализированных потомках `TStream` определены стандартные методы `Read`, `Write` и `Seek`, открывающие полный доступ к файловому объекту. Сам объект создается конструктором и уничтожается деструктором.

ВНИМАНИЕ Поток не учитывает специфику хранящихся в файле данных, то есть, по существу, работает с данными как если бы они хранились в нетипизированном файле.

В следующем примере файл с именем `Edit1.Text` копируется в файл `Edit2.Text`.

```
procedure TForm1.CopyFileClick(Sender:TObject);
var
    Stream1, Stream2: TStream;
begin
    Stream1:=TFileStream.Create(Edit1.Text,
fmOpenRead or fmShareDenyWrite);
    try
        Stream2 :=TFileStream.Create(Edit2.Text,
            fmOpenCreate or fmShareDenyRead);
        try
            Stream2.CopyFrom(Stream1, Stream1.Size);
        finally
            Stream2.Free;
        finally
            Stream1.Free
    end;
```

Абстрактный класс `TStream` лишь декларирует ключевые методы `Read` и `Write`, которые перекрываются в его наследниках для специализации операций:

- ✱ `TFileStream` — обмен данными с файлом;
- ✱ `TStringStream` — обмен данными с длиной строкой (в памяти);
- ✱ `TMemoryStream` — обмен данными с памятью;
- ✱ `TBLOBStream` — чтение/запись полей данных типа BLOB (Binary Large Object — большой двоичный объект);
- ✱ `TWinSocketStream` — обмен данными с сокетным подключением;
- ✱ `TOLEStream` — обмен данными в формате OLE.

Любой наследник получает в свое распоряжение метод `CopyFrom`, с помощью которого он может прочитать содержимое другого потока. Таким способом можно, например, файловые данные расположить в памяти или наоборот — содержимое памяти записать в дисковый файл.

Точно так же от TStream наследуются множество других полезных методов, в том числе:

- Seek — пропускает нужное количество байтов от текущей позиции потока;
- ReadBuffer — читает из буфера в поток нужное количество байтов;
- ReadComponent — читает свойства компонента из потока;
- WriteBuffer — записывает в поток содержимое буфера заданной длины;
- WriteComponent — записывает в поток свойства компонента.

Следующий обработчик bbRunClick выведет в поле mmOutput свойства кнопки bbRun (Source\Chap_10\WriteComp\WriteComp.dpr):

```
procedure TfmExample.bbRunClick(Sender: TObject);
var
    MemSourceStream, MemDestStream: TMemoryStream;
begin
    MemSourceStream := TMemoryStream.Create;
    try
        MemDestStream := TMemoryStream.Create;
        try
            MemSourceStream.WriteComponent(bbRun);
            MemSourceStream.Seek(0, soFromBeginning);
            ObjectBinaryToText(MemSourceStream, MemDestStream);
```

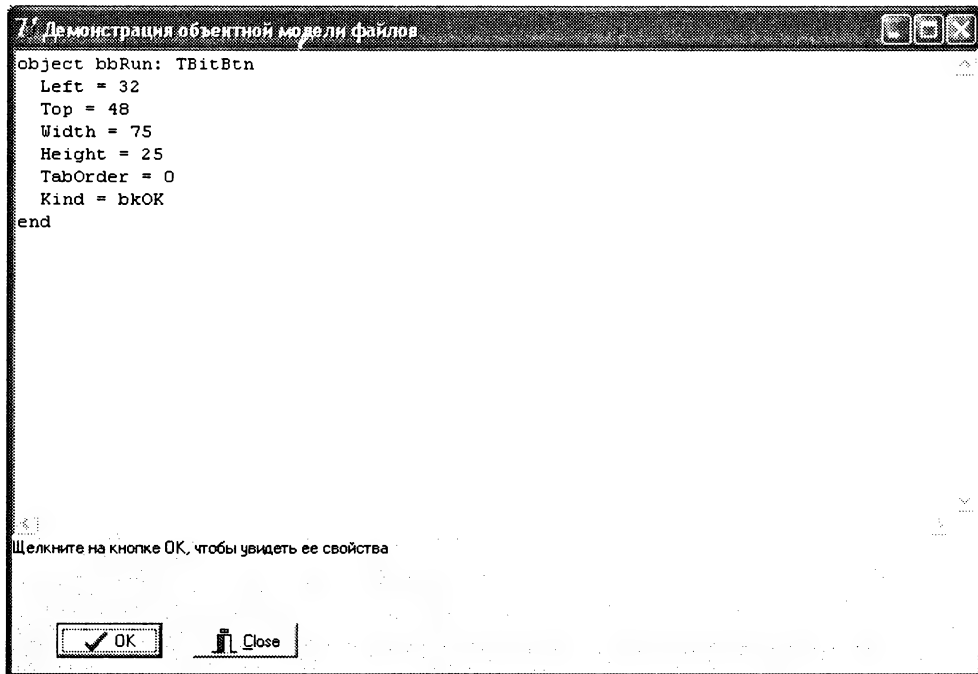


Рис. 10.3. Свойства компонента bbRun

```
        MemDestStream.Seek(0, soFromBeginning);  
        mmOutput.Lines.LoadFromStream(MemDestStream)  
    finally  
        MemDestStream.Free  
    end;  
finally  
    MemSourceStream.Free  
end;  
end;
```

Вначале свойства компонента записываются в поток `MemSourceStream`. Чтобы полученным таким образом двоичным данным придать «читабельный» вид, используется процедура `ObjectBinaryToText`, которая преобразует данные из `MemSourceStream` и помещает их в `MemDestStream`. Содержимое этого второго потока и выводится в окне `mmOutput`.

Вид окна работающей программы представлен на рис. 10.3.

Глава 11

Модули

Стандартный язык Паскаль не предусматривает механизмов отдельной компиляции частей программы с последующей их сборкой перед выполнением. Более того, последовательное проведение в жизнь принципа обязательного описания любого объекта перед его использованием делает фактически невозможным разработку разнообразных библиотек прикладных программ. Точнее, такие библиотеки в рамках стандартного языка Паскаль могут существовать только в виде исходных текстов, и программист должен сам включать в программу подчас весьма обширные тексты различных поддерживающих процедур, таких, как процедуры матричной алгебры, численного интегрирования, математической статистики и т. п.

Вполне понятно поэтому стремление разработчиков коммерческих компиляторов Паскаля включать в язык средства, повышающие его модульность. Чаще всего таким средством является разрешение использовать внешние процедуры и функции, тело которых заменяется стандартной директивой **External**. Разработчики Delphi пошли в этом направлении еще дальше, включив в язык механизм так называемых модулей.

Модуль — это автономно компилируемая программная единица, включающая в себя различные компоненты интерфейсного раздела (типы, константы, переменные, процедуры и функции) и, возможно, некоторые исполняемые операторы инициализирующего раздела. Появление объектов в интерфейсной части делает их доступными для других модулей и основной программы. Тела процедур и функций располагаются в исполняемой части модуля, которая может быть скрыта от пользователя.

Роль модулей в Delphi не исчерпывается только механизмом отдельной компиляции. Delphi связывает с каждым включаемым в программу окном собственный модуль и таким способом локализует все свойства окна в отдельной программной единице. Как мы выясним далее (см. главу 22), основная программа (DPR-файл проекта) весьма специфична: она содержит перечень всех используемых в программе модулей и несколько исполняемых операторов, обеспечивающих создание нужных окон и связь программы с Windows. Вся основная работа программы управляется кодом, содержащимся в модулях.

Структура модулей

Модуль имеет следующую структуру:

```
Unit <имя>;  
interface  
<интерфейсная часть>  
implementation  
<исполняемая часть>  
initialization  
<инициализирующая часть>  
finalization  
<завершающая часть>  
end.
```

Здесь **Unit** — зарезервированное слово (единица), начинающее заголовок модуля; <имя> — имя модуля (правильный идентификатор); **interface** — зарезервированное слово (интерфейс), начинающее интерфейсную часть модуля; **implementation** — зарезервированное слово (выполнение), начинающее исполняемую часть; **initialization** — зарезервированное слово (инициализация), начинающее инициализирующую часть модуля; **finalization** — зарезервированное слово (завершение), начинающее завершающую часть модуля; **end** — зарезервированное слово, являющееся признаком конца модуля.

Таким образом, модуль состоит из заголовка и четырех составных частей, любая из которых может быть пустой.

Заголовок модуля и связь модулей друг с другом

Заголовок модуля состоит из зарезервированного слова **Unit** и следующего за ним имени модуля. Для правильной работы среды Delphi и возможности подключения средств, облегчающих разработку крупных программ, это имя должно совпадать с именем дискового файла, в который помещается исходный текст модуля. Например, пусть имеем заголовок:

```
Unit Global;
```

Тогда исходный текст соответствующего модуля должен размещаться в дисковом файле GLOBAL.PAS. Имя модуля служит для его связи с другими модулями и основной программой. Эта связь устанавливается специальным предложением:

```
Uses <сп.модулей>
```

Здесь **Uses** — зарезервированное слово (использует); <сп.модулей> — список модулей, с которыми устанавливается связь; элементами списка являются имена модулей, отделяемые друг от друга запятыми, например:

```
Uses Windows, SysUtils, MyUnit;
```

Если объявление **Uses** используется, оно должно открывать раздел описаний основной программы. Модули могут использовать другие модули. Предложение

Uses в модулях может следовать либо сразу за зарезервированным словом **interface**, либо сразу за словом **implementation**, либо, наконец, там и там (то есть в модуле допускаются два предложения **Uses**).

Интерфейсная часть

Интерфейсная часть открывается зарезервированным словом **interface**. В этой части содержатся объявления всех глобальных объектов модуля (типов, констант, переменных и подпрограмм), которые должны стать доступными основной программе и/или другим модулям. При объявлении глобальных подпрограмм в интерфейсной части указывается только их заголовок, например:

```
Unit Cmplx;
interface
type
  Complex = record
    re,im: Real
  end;
function AddC(x,y: Complex): Complex;
function MulC(x,y: Complex): Complex;
```

Если теперь в другом модуле написать показанное ниже предложение, то в нем станут доступными тип **Complex** и две процедуры — **AddC** и **MulC** из модуля **Cmplx**:

```
Uses Cmplx;
```

Исполняемая часть

Исполняемая часть начинается зарезервированным словом **implementation** и содержит описания подпрограмм, объявленных в интерфейсной части. В ней могут объявляться локальные для модуля объекты — вспомогательные типы, константы, переменные и подпрограммы, а также метки, если они используются в инициализирующей части.

Описанию подпрограммы, объявленной в интерфейсной части модуля, в исполняемой части должен предшествовать заголовок, в котором можно опускать список формальных переменных (и тип результата для функции), так как они уже описаны в интерфейсной части. Но если заголовок подпрограммы приводится в полном виде, то есть со списком формальных параметров и объявлением результата, он должен совпадать с заголовком, объявленным в интерфейсной части, например:

```
Unit Cmplx;
Interface
type
  Complex = record
    re,im: real
  end;
function AddC(x,y: Complex): Complex;
function MulC(x,y: Complex): Complex;
```

Implementation

```

function AddC(x, y: Complex): Complex;
begin
    ...
end;

function MulC; // Вариант описания подпрограммы без
               // повторения списка параметров
begin
    ...
end;
end.

```

СОВЕТ Хотя и допускается краткое объявление заголовка подпрограммы (как в предыдущем примере — функции `MulC`), использовать такую форму в серьезной программе не рекомендуется: перечень параметров непосредственно в заголовке подпрограммы облегчает чтение кода и понимание деталей реализации алгоритма. Для точного указания заголовка подпрограммы в исполняемом разделе используйте возможности редактора кода: установите курсор в любое место заголовка подпрограммы в интерфейсной части и нажмите клавиши `Ctrl+Shift+C` — кодовый редактор создаст заготовку подпрограммы в исполняемой части модуля.

Повторение заголовка в исполняемой части должно быть полным и точным. Если бы мы использовали следующий заголовок, компилятор немедленно извещал бы нас о несовпадении заголовка с объявлением функции в интерфейсной части (второй параметр должен иметь имя `y`):

```

function AddC(x, z: Complex): Complex;
begin
    ...
end;

```

Инициализирующая и завершающая части

Инициализирующая и завершающая части чаще всего отсутствуют вместе с начинающим их словами **initialization** и **finalization**.

В инициализирующей части размещаются операторы, которые исполняются *до* передачи управления основной программе и обычно используются для подготовки ее работы. Например, в них могут инициализироваться переменные, открываться нужные файлы и т. д. В завершающей части указываются операторы, выполняющиеся *после* завершения работы основной программы (в них освобождаются выделенные программе ресурсы, закрываются файлы и т. д.). Если несколько модулей содержат инициализирующие части, эти части выполняются последовательно друг за другом в порядке перечисления модулей в предложении **Uses** главной программы. Если несколько модулей содержат завершающие части, эти части выполняются последовательно друг за другом в порядке, обратном перечислению модулей в предложении **Uses** главной программы.

Доступ к объявленным в модуле объектам

Пусть, например, мы создаем модуль, реализующий арифметику комплексных чисел (такая арифметика ни в стандартном языке Паскаль, ни в Delphi не предусмотрена, но в Delphi введен пользовательский вариант, который реализует действия над комплексными числами — см. модуль Source\Rtl\Common\VarCmplx.pas каталога размещения Delphi). Арифметика комплексных чисел реализуется четырьмя функциями:

```
Unit Cmplx;
//-----

                        Interface
//-----
type
    Complex = record
                re,im: real
            end;

function AddC (x,y: Complex): Complex;
function SubC (x,y: Complex): Complex;
function MulC (x,y: Complex): Complex;
function DivC (x,y: Complex): Complex;

const
    c : Complex = (re : 0.1; im : -1);
//-----

                        Implementation
//-----
function AddC (x,y: Complex): Complex;
// Сложение комплексных чисел
begin
    Result.re := x.re + y.re;
    Result.im := x.im + y.im
end; //AddC

function SubC (x,y: Complex): Complex;
// Вычитание комплексных чисел
begin
    Result.re := x.re - y.re;
    Result.im := x.im - y.im
end; // SubC

function MulC (x,y: Complex): Complex;
// Умножение комплексных чисел
begin
    Result.re := x.re * y.re - x.im * y.im;
```

```
    Result.im := x.re * y.im + x.im * y.re
end; // MulC

function DivC (x,y: Complex): Complex;
// Деление комплексных чисел
var
    z: Real;
begin
    z := sqr(y.re) + sqr(y.im);
    // Защищаем программу от краха в случае, когда z=0:
    try
        Result.re := (x.re * y.re + x.im * y.im) / z;
        Result.im := (x.re * y.im - x.im * y.re) / z;
    except
        Result.re := 1.1e309;
        Result.im := 1.1e309;
    end
end {DivC};

end.
```

Чтобы создать такой модуль, следует выбрать команду File ► New ► Unit. Текст модуля следует сохранить в файле CMPLX.PAS; *имя файла должно совпадать с именем модуля* — только в этом случае Delphi сможет автоматически найти модуль и следить за его обновлением.

После создания модуля его имя нужно упомянуть в предложении **Uses** того модуля, в котором будут использоваться вновь созданные подпрограммы, типы, константы (в нашем модуле — тип Complex, подпрограммы AddC, SubC, MulC, DivC и константа C). Пусть, например, при каждом щелчке на кнопке bbRun учебной программы создается пара случайных комплексных чисел, над которыми осуществляются все четыре арифметических действия. Тогда обработчик bbRunClick мог бы быть таким (проект Source\Chap_11\Complex\Complex.dpr):

```
implementation
```

```
uses Cmplx;
```

```
{ $R *.DFM }
```

```
procedure TForm1.bbRunClick(Sender: TObject);
```

```
var
```

```
    x, y, z: Complex;
```

```
procedure Output(Operation: Char);
```

```
// Осуществляет нужное действие и выводит результат
```

```
// в mmOutput
```

```
var
```

```

    S: String;
begin
    case Operation of
        '+': z := AddC(x,y);
        '-': z := SubC(x,y);
        '*': z := MulC(x,y);
        '/': z := DivC(x,y);
    end;
    S := ('+FormatFloat('+0.0000;-0.0000',x.re)+
        FormatFloat('+0.0000j;-0.0000j',x.im)+')'+
        Operation+
        ('+FormatFloat('+0.0000;-0.0000',y.re)+
        FormatFloat('+0.0000j;-0.0000j',y.im)+'='+
        FormatFloat('+0.0000;-0.0000',z.re)+
        FormatFloat('+0.0000j;-0.0000j',x.im));
    mmOutput.Lines.Add(S);
end; // Output

begin // bbRunClick
    x.re := Random;
    x.im := Random;
    y.re := Random;
    y.im := Random;
    Output('+');
    Output('-');
    Output('*');
    Output('/');
    mmOutput.Lines.Add('');
end;

```

Обратите внимание на ссылку **uses** Cmplx в самом начале исполняемой части — именно она делает доступными обработчику bbRunClick объекты модуля Cmplx. Эту ссылку можно вставить с помощью среды Delphi: выберите команду File ► Use Unit главного меню и в появившемся окне щелкните на имени модуля Cmplx.

Типы модулей в Delphi

Наиболее распространенным типом модуля в Delphi является *форма* — модуль со связанным с ним окном. Интерфейсная часть такого модуля обычно содержит объявление нового класса и автоматически обновляется Delphi в ходе конструирования окна. В интерфейсной части модуля-формы содержится также объявление объекта для соответствующего оконного класса. Например, для нашей учебной программы модуль содержит объявление класса TfmExample и объекта fmExample. Большинство типовых модулей в хранилище объектов содержат заготовки для создания диалоговых окон.

Помимо форм в хранилище содержатся также не связанные с видимыми окнами модули. Кроме уже рассмотренного выше модуля общего назначения, к ним относятся модули данных, модули динамических библиотек, пакеты и модули потоков.

Модули данных

Модули данных имеют связанные с ними окна, однако эти окна никогда не появляются на экране. Необходимость в окнах вызвана тем, что компоненты доступа к данным, стандартные диалоговые окна, таймер и некоторые другие компоненты можно вставить только в форму, хотя все они не имеют видимого воплощения в работающей программе. Невидимое окно модуля данных предназначено для размещения подобных компонентов и связанных с ними объектов-полей. Разумеется, для размещения компонентов и полей можно использовать и обычное окно-форму, однако в этом случае значки компонентов загромождают видимое пространство окна и затрудняют его конструирование. В Delphi 7 модули данных способны отображать реляционные связи между сущностями базы данных в виде диаграмм.

Модули динамических библиотек

Модули динамических библиотек предназначены для создания широко используемых в Windows динамически подключаемых библиотек (Dynamic-Link Libraries, DLL). DLL служат универсальным средством согласования подпрограмм, написанных на разных языках программирования. В Windows содержится множество DLL, написанных на языке Си или на языке ассемблера, что ничуть не мешает Delphi-программам использовать их. Модули динамических библиотек предназначены для разработки DLL с помощью Delphi. Такие DLL затем смогут использовать программы, созданные с помощью других языков программирования. Методика создания DLL рассматривается в главе 23.

Пакеты

Пакеты — это особым образом откомпилированные библиотеки DLL, оптимизированные для совместного использования Delphi-программами или средой Delphi, или и программами и средой. В отличие от DLL, пакеты могут хранить и передавать программе типы (включая классы) и данные. Они разработаны специально для хранения компонентов, разного рода экспертов, редакторов сложных свойств и т. п. Например, в пакете VCL70.BPL содержатся основные компоненты Delphi.

Модули потоков команд

Модули потоков предназначены для реализации так называемых потоков команд — фрагментов программы, которые исполняются параллельно с другими фрагментами, разделяя с ними время процессора и остальные системные ресурсы. Механизм потоков команд используется в 32-разрядных версиях Windows и не поддерживается в Delphi 1. К сожалению, в текущей реализации 32-разрядной версии Delphi потоки команд не могут связываться с собственными видимыми компонентами, так как библиотека визуальных компонентов (Visual Component Library, VCL) не поддерживает работу с потоками. Вот почему модуль потока не имеет связанного с ним окна. Методика использования потоков рассматривается в главе 22.

Глава 12

Типизированные константы и инициализированные переменные

В этой небольшой главе рассматривается методика объявления типизированных констант и инициализированные переменные. Объединение констант и переменных в одной главе не случайно: как уже говорилось, типизированные константы могут изменяться в программе и, следовательно, фактически являются инициализированными переменными.

Типизированные константы

В Delphi допускается использование типизированных констант. Они задаются в разделе объявления констант следующим образом:

```
<идентификатор>: <тип> = <значение>
```

Здесь <идентификатор> — идентификатор константы; <тип> — тип константы; <значение> — значение константы.

Типизированным константам можно присваивать другие значения в ходе выполнения программы¹, поэтому фактически они представляют собой переменные с начальными значениями, то есть являются *инициализированными* переменными. Типизированная константа приобретает указанное в ее объявлении значение, то есть инициализируется, лишь один раз: к моменту начала работы программы. При повторном входе в блок (процедуру или функцию), в котором она объявлена, инициализация типизированной константы не производится и она сохраняет то значение, которое имела к моменту выхода из блока.

¹ В Delphi 7 присваивание новых значений типизированным переменным по умолчанию запрещено. Чтобы снять запрет, нужно установить флажок `Assignable typed constants` на вкладке `Compiler` диалогового окна `Options` (открывается командой `Project ► Options`).

Типизированные константы могут быть любого типа, кроме файлов, объектов и классов. Нельзя также объявить типизированную константу-запись, если хотя бы одно из ее полей является полем файлового типа, объектом или классом.

Поскольку типизированная константа фактически не отличается от переменной, ее нельзя использовать в качестве значения при объявлении других констант или границ типа-диапазона.

Константы простых типов и типа String

Объявление констант простых типов и типа **String** обычно не вызывает трудностей, так как в качестве их значения используются нетипизированные константы или их идентификаторы.

Примеры объявлений:

type

```
colors = (white, red, black);
```

```
{ ----- Правильные объявления: ----- }
```

const

```
CurrCol : colors = red;
```

```
Name : String = 'Вирт Н.';
```

```
Year : Word = 1989;
```

```
X : Real = 0.1;
```

```
Min : Integer = 0;
```

```
Max : Integer = 10;
```

```
Days : 1..31 = 1;
```

```
Answer : Char = 'Y';
```

```
{ ----- Неправильные объявления: ----- }
```

```
Mass : array [Min..Max] of Real; {Нельзя использовать  
типизированные константы в качестве границ диапазона}
```

```
a,b,c: Byte = 0; {Нельзя использовать список  
идентификаторов}
```

Строковые константы можно размещать в так называемых ресурсах — файлах специального типа, которые присоединяются к программе на этапе сборки исполняемого файла. Для объявления строкового ресурса используется зарезервированное слово **resourcestring**. Например:

resourcestring

```
R1 = 'Строка 1';
```

```
R2 = 'Строка 2';
```

Строковые ресурсы в программе используются обычным образом:

var

```
S: String;
```

begin

```
S := R1 + R2;
```

```
R1 := ' ' // Ошибка! Константе нельзя присвоить значение!
```

end;

Константы-массивы

В качестве начального значения типизированной константы-массива используется список констант, отделенных друг от друга запятыми; список заключается в круглые скобки, например:

```
type
  colors = (white, red, black);
const
  ColStr : array [colors] of String [5] =
      ('white', 'red', 'blak');
  Vector : array [1..5] of Byte = (0,0,0,0,0);
```

При объявлении массива символов можно использовать то обстоятельство, что все символьные массивы и строки в Delphi хранятся в упакованном формате, поэтому в качестве значения массива-константы типа **Char** допускается задание символьной строки соответствующей длины. Два следующих объявления идентичны:

```
const
  Digit : array [0..9] of Char =
      ('0','1','2','3','4','5','6','7','8','9');
  DigChr: array [0..9] of Char = '0123456789';
```

При объявлении многомерных констант-массивов множество констант, соответствующих каждому измерению, заключается в дополнительные круглые скобки и отделяется от соседнего множества запятыми. В результате образуются вложенные структуры множеств, причем глубина вложения должна соответствовать количеству измерений (размерности) массива. Самые внутренние множества констант связываются с изменением самого правого индекса массива.

Следующая программа выведет в компонент **mmOutput** три строки с монотонно увеличивающимися целыми числами:

```
Procedure TfmExample.bbRunClick(Sender: TObject);
var
  i, j, k, l : Byte;
  S: String;
const
  Matr : array [1..3, 1..5] of Byte =
      ((0, 1, 2, 3, 4),
       (5, 6, 7, 8, 9),
       (10,11,12,13,14));
  Cube : array [0..1, 0..1, 0..2] of Integer =
      (((0 ,1 ,2 ), (3 ,4 ,5 )),
       ((6 ,7 ,8 ), (9 ,10,11)));
  Mas4 : array [0..1, 0..1, 0..1, 0..1] of Word =
      ((( (0 ,1 ), (2 ,3 )),
        ((4 ,5 ), (6 ,7 ))),
       (( (8 ,9 ), (10,11)),
        ((12,13), (14,15)))));
```

```

begin
    S := '';
    for i := 1 to 3 do
        for j := 1 to 5 do
            S := S + IntToStr(Matrx[i,j]) + ' ';
        mmOutput.Lines.Add(S);
    S := '';
    for i := 0 to 1 do
        for j := 0 to 1 do
            for k := 0 to 2 do
                S := S + IntToStr(Cube[i,j,k]) + ' ';
            mmOutput.Lines.Add(S);
        S := '';
        for i := 0 to 1 do
            for j := 0 to 1 do
                for k := 0 to 1 do
                    for l := 0 to 1 do
                        S := S + IntToStr(mas4[i,j,k,l])+ ' ';
                    mmOutput.Lines.Add(S);
                end;
            end;
        end;
    end;
end;

```

Количество переменных в списке констант должно строго соответствовать объявленной длине массива по каждому измерению.

Константы-записи

Определение константы-записи имеет следующий вид:

<идентификатор>: <тип> = (<сп.знач.полей>)

Здесь <идентификатор> — идентификатор константы; <тип> — тип записи; <сп.знач.полей> — список значений полей.

Список значений полей представляет собой список из последовательностей вида: имя поля, двоеточие и константа. Элементы списка отделяются друг от друга двоеточиями, например:

```

type
    Point = record
        X, Y : Real
    end;
    Vect  = array [0..1] of Point;
    Month = (Jan, Feb, Mar, Apr, May, Jun,
            Jly, Aug, Sep, Oct, Nov, Dec);
    Date  = record
        D : 1..31;
        M : month;
        Y : 1900..1999
    end;
const
    Origin : Point = (X : 0; Y : -1);

```

```
Line : Vector = ((X : -3.1; Y : 1.5), (X : 5.9; Y : 3.0));
SomeDay : Date = (D : 16; M : May; Y : 1997);
```

Поля должны указываться в той последовательности, в которой они перечислены в объявлении типа. Если в записи используется хотя бы одно поле файлового типа, такую запись нельзя объявить типизированной константой. Для записей с вариантными полями указывается только один из возможных вариантов констант. Например:

```
type
  Forma = record
    case Boolean of
      True   : (BirthPlace: String [40]);
      False  : (Country   : String [20];
                EntryPort : String [20];
                EntryDate : array [1..3] of Word;
                Count     : Word)
    end;
const
  Percon1 : Forma = (Country   : 'Норвегия';
                     EntryPort : 'Мурманск';
                     EntryDate : (16, 3, 89);
                     Count     : 12);
  Percon2 : Forma = (BirthPlace: 'Москва');
```

Константы-множества

Значение типизированной константы-множества задается в виде правильного конструктора множества, например:

```
type
  Days   = set of 1..31;
  Digs   = set of '0'..'9';
  Error  = set of 1..24;
const
  WorkDays : Days = [1..5, 8..12, 12..19, 22..26, 29, 30];
  EvenDigits: Digs = ['0', '2', '4', '6', '8'];
  ErrorFlag : Error= [];
```

Константы-указатели

Единственным значением типизированной константы-указателя может быть только **NIL**, например:

```
const pR: ^Real = NIL;
```

Инициализация переменных

Во всех 32-разрядных версиях Delphi при объявлении глобальных переменных разрешается одновременно присваивать им значения (инициализировать). Речь идет о *глобальных* переменных, объявляемых в интерфейсной или исполняемой

частях модулей (переменные, объявляемые в теле процедуры, называются локальными, их инициализировать нельзя; вместо этого используйте описанные выше локальные типизированные константы).

Инициализированная переменная получает присвоенное ей значение один раз — в момент старта программы; в дальнейшем она хранит последнее присвоенной ей значение. Пример инициализации переменной:

```
unit Unit1;  
...  
implementation  
var  
    MyVar: Integer = 10;  
...  
end.
```

Часть III

Компоненты и программа

В этой части описываются некоторые наиболее часто используемые классы, компоненты и программа в целом

Глава 13

Классы общего назначения

В состав Delphi входит около 400 различных стандартных классов, простое перечисление которых заняло бы несколько страниц книги. В этой главе рассматриваются лишь некоторые самые важные классы общего назначения.

Класс Exception — обработка исключений

Класс `Exception` является прямым потомком базового класса `TObject`. Вместе со своими потомками он предназначен для обработки исключительных ситуаций (исключений), возникающих при некорректных действиях программы: например, в случае деления на ноль, при попытке открыть несуществующий файл, при выходе за пределы выделенной области динамической памяти и т. п. В этом разделе рассматриваются основные свойства исключений и их использование для повышения надежности программ.

СОВЕТ При работе в среде Delphi эксперименты с исключениями плохо прослеживаются, так как при каждом исключении среда перехватывает управление программой. В этом случае бывает полезно отменить такое поведение среды. Для этого откройте диалоговое окно `Debugger Options`, выбрав команду `Tools ▶ Debugger Options`, и на вкладке `Language Exceptions` снимите флажок `Stop on Delphi Exceptions`.

Защищенные блоки

Для обработки исключений в Delphi предусмотрены два типа защищенных блоков:

```
try
  <операторы>
except
  <обработчики исключений>
else
  <операторы>
end;
```

```

try
    <операторы>
finally
    <операторы>
end;

```

Защищенный блок начинается зарезервированным словом **try** (попытаться [выполнить]) и завершается словом **end**. Два типа защищенных блоков **except** (исключить) и **finally** (в завершение) отличаются способом обработки исключения. В блоке **except** порядок выполнения операторов таков: сначала выполняются операторы секции **try...except**; если операторы выполнены без возникновения исключительной ситуации, работа защищенного блока на этом прекращается и управление получает оператор, стоящий за словом **end**; если при выполнении части **try** возникло исключение, управление получает соответствующий обработчик в секции **except**, а если таковой не найден — первый из операторов, стоящих за словом **else**. В блоке **finally** операторы в секции **finally...end** получают управление всегда, независимо от того, возникло исключение в секции **try...finally** или нет. Если исключение возникло, все операторы в секции **try...finally**, стоящие за «виновником» исключения, пропускаются и управление получает первый оператор секции **finally...end**. Если исключения не было, этот оператор получает управление после выполнения последнего оператора секции **try...finally**.

Обработчики исключений в блоке **except** имеют такой синтаксис:

```

on <класс исключения> do <оператор>;

```

Здесь **on**, **do** — зарезервированные слова; <класс исключения> — класс обработки исключения; <оператор> — любой оператор Delphi, *кроме оператора передачи управления goto* на метку вне блока **except**.

Обратите внимание: имя класса служит своеобразным ключом выбора, а собственно обработка осуществляется оператором, стоящим за **do** (этот оператор бывает составным, так что обработка исключения может выполняться произвольным количеством операторов Delphi).

Поиск нужного обработчика осуществляется с начала списка вниз до тех пор, пока не встретится класс, способный обрабатывать исключение данного типа. Если подходящего класса не обнаружено, управление передается операторам, стоящим за словом **else**, а если таковых нет (часть **else** <операторы> может опускаться), выполняется умалчиваемая обработка исключения.

Если для программиста важен лишь сам факт возникновения исключения и не существен тип связанной с ним ошибки, он может опустить в секции **except...end** обработчики вместе со словом **else**, оставив в ней лишь необходимый код реакции на любую ошибку:

```

try
    ...
except
    ShowMessage ('Ошибка!');
    ...
end;

```

Защищенные блоки могут вкладываться друг в друга на неограниченную глубину, так как везде, где в предыдущих описаниях указывался параметр <оператор>, могут использоваться любые операторы Delphi, в том числе и **try...except** или **try...finally**:

```

try
    ...
    try
        ...
    finally
        ...
    end;
except
    on EMatchError do
    begin
        ...
        try
            ...
            try
                ...
            end;
            ...
        end;
        ...
    end;
    ...
end;
end;

```

Класс Exception

Класс Exception является родительским классом для всех классов-исключений. Этот класс объявляется в модуле SysUtils следующим образом:

```

type
    Exception = class(TObject)
    private
        FMessage: String;
        FHelpContext: Integer;
    public
        constructor Create(const Msg: String);
        constructor CreateFmt(const Msg: String;
            const Args: array of const);
        constructor CreateRes(Ident: Integer);
        constructor CreateResFmt(Ident: Integer;
            const Args: array of const);
        constructor CreateHelp(const Msg: String; aHelpContext:
            Integer);
        constructor CreateFmtHelp(const Msg: String;

```



```

    const Args: array of const; aHelpContext: Integer);
    constructor CreateResHelp(Ident: Integer; aHelpContext:
        Integer);
    constructor CreateResFmtHelp(Ident: Integer;
        const Args: array of const; aHelpContext: Integer);
    property HelpContext: Integer read
        FHelpContext write FHelpContext;
    property Message: String read FMessage
        write FMessage;
end;

```

В классе определены целых 8 конструкторов для создания объекта. С их помощью можно прочитать текстовое сообщение из ресурса, отформатировать его, связать исключение с контекстной справочной службой. Свойство `Message` делает доступным закрытое поле `FMessage`, в котором содержится текстовое сообщение.

Стандартные классы исключений

В Delphi определены стандартные классы исключений (см. приложение Б). Именно эти имена, а также имена пользовательских классов (см. ниже) могут применяться в обработчиках исключений.

Важно помнить, что управление передается самому первому обработчику (возможно существование нескольких обработчиков), класс которого способен обрабатывать данное исключение. Если, например, в списке первым стоит класс `EAbort`, который может обработать любое исключение, ни один из стоящих за ним обработчиков никогда не получит управления. Точно также, если указан обработчик для класса `EIntError`, за ним бесполезно размещать обработчики `EDivByZero`, `ERangeError` или `EIntOverflow`:

```

try
    ...
except
    // Не имеет смысла делать так:
    on EIntError do ...;
    on ERangeError do ...;
    on EDivByZero do ...;
    // Надо так:
    on ERangeError do ...;
    on EDivByZero do ...;
    on EIntError do ...;
end;

```

При возникновении исключительной ситуации объекты классов-обработчиков создаются и уничтожаются автоматически. Если программист пожелает использовать поля или методы класса-обработчика явно, он должен поименовать автоматически создаваемый объект. Для этого перед именем класса ставится идентификатор и двоеточие:

```

on EObject: EClassName do ...;

```

Для стандартных классов такой прием фактически позволяет использовать единственное строковое свойство `Message` со стандартным сообщением об ошибке, которое получают все наследники класса `Exception`. Исключение составляет класс `EInOutError`, в котором для программиста может представлять интерес целочисленное свойство `ErrorCode` с кодом ошибки ввода-вывода. Например:

```
try
    Reset (F) ;
    while not EOF (F) do
    begin
        ...
    end;
    CloseFile (F) ;
except
    on E: EInOutError do
        ShowMessage ('При выполнении файловой операции возникла '
            + ' ошибка №'+ IntToStr (E.ErrorCode));
end;
```

Вызов исключения

В некоторых ситуациях программисту бывает необходимо инициировать собственное исключение. Для этого он использует зарезервированное слово **raise** (возбудить). Если это слово встретилось в секции **try...exception** или **try...finally**, немедленно начинают свою работу соответственно секция **except...end** или **finally...end**. Если оно встретилось в секции **except...end** или **finally...end**, считается, что данный защищенный блок на текущем уровне вложенности (блоки могут быть вложенными) завершил свою работу и управление передается вышестоящему уровню.

Слово **raise** возбуждает исключение самого общего класса `Exception`. Если программист желает возбудить исключение конкретного типа (неважно, стандартного или собственного), он должен явно указать класс создаваемого в этот момент объекта путем вызова его конструктора. Например, следующий оператор возбудит ошибку ввода-вывода:

```
raise EInOutError.Create ('Ошибка!');
```

Такой прием — единственная возможность возбудить нестандартное исключение, обрабатываемое пользовательским классом.

Создание собственного класса

Программист может создать собственный класс обработки исключений, объявив его потомком `Exception` или любого другого стандартного класса (этим другим чаще всего бывает класс `EAbort`). Объявление нестандартного класса имеет смысл только тогда, когда вам необходимо научить программу распознавать некорректные наборы данных и соответствующим образом на них реагировать.

Пусть, например, в программе используется цикл ввода целочисленных значений из текстового файла, их проверки и преобразования. Проверка заключается в простом контроле неотрицательности очередного числа после ввода и его поло-

жительности после преобразования. Перед проверкой необходимо получить строку из файла (здесь может возникнуть ошибка `EInOutError`) и преобразовать ее в целую величину (здесь возможна ошибка `EConvertError`); после проверки осуществляется обработка величины, в процессе которой может возникнуть ошибка `EIntError`.

Создадим новый класс `EIntCheckError` и будем возбуждать исключение этого класса при обнаружении ошибки в данных:

```
type
  EIntCheckError = class (EAbort)
end;

var
  F: TextFile;
  S: String;
  k: Integer;
begin
  ...
  try
    // Готовимся к работе: открываем файл
    AssignFile(F, FileName);
    Reset(F);           // Здесь возможна ошибка EInOutError
    // Цикл ввода-контроля-преобразования
    while not EOF(F) do
      begin
        // Вводим символы очередного числа
        ReadLn(F,S);     // Здесь возможна ошибка EInOutError
        // Преобразуем символы в число
        k := StrToInt(S); // Здесь возможна ошибка EConvertError
        // Проверяем число
        if k < 0 then
          raise EIntCheckError.Create('Отрицательное число');
        // Преобразуем число
        ...              // Здесь возможна ошибка EIntError
        // Вновь проверяем число
        if k <= 0 then
          raise EIntCheckError.Create('Не положительное
            число');
        ...
      end;
    except
      on E: EIntCheckError do
        ShowMessage(E.Message);
      on EInOutError do
        ShowMessage('Некорректная файловая операция');
      on EConvertError do
        ShowMessage('Ошибка в записи числа');
```

```

on EIntError do
    ShowMessage('Ошибка преобразования');
end;
end;

```

В этом примере создается класс EIntCheckError, который ничем, кроме названия, не отличается от своего родителя EAbort. В реальной программе потомок обычно расширяет набор полей (свойств) своего родителя или перекрывает его методы; приведенный пример лишь иллюстрирует, что этого делать *не обязательно*. При неудачной проверке следующими операторами возбуждается исключение нового класса:

```

raise EIntCheckError.Create('Отрицательное число');
raise EIntCheckError.Create('Ошибка преобразования');

```

При этом с помощью унаследованного конструктора Create создается новый безымянный объект, а строковый параметр обращения к конструктору запоминается в поле FMessage и становится доступным с помощью свойства Message объекта. Обработчик исключения EIntCheckError именуется объект идентификатором E и с помощью стандартной процедуры ShowMessage показывает его в небольшом окне на экране.

Пример наглядно иллюстрирует выгоды от использования исключений. В принципе, весь этот фрагмент можно было бы написать с многочисленными проверками **if...then**, но в этом случае логика программы стала бы запутанной, а сама программа — сложной в отладке.

Класс TList — списки

Класс TList позволяет создать набор из произвольного количества элементов и организовать индексный способ доступа к ним, как это делается при работе с массивом. Списки отличаются от массивов двумя важными особенностями. Во-первых, их размер может динамически меняться в ходе работы программы, фактически ограничиваясь лишь доступной памятью. Во-вторых, в списках могут храниться элементы *разных* типов.

Технически списки представляют собой массивы нетипизированных указателей на размещенные в динамической памяти элементы. Эти массивы размещаются в куче — отсюда возможность динамического изменения размеров списков; с другой стороны, входящие в списки *нетипизированные* указатели позволяют ссылаться на произвольные элементы.

В табл. 13.1 указаны наиболее важные свойства класса.

Таблица 13.1. Свойства класса TList

Свойство	Описание
property Capacity: Integer;	Содержит количество элементов массива указателей списка. Всегда больше Count. Если при добавлении очередного элемента значение Count стало равно Capacity, происходит автоматическое расширение списка на 16 элементов

— продолжение ➤

Таблица 13.1 (продолжение)

Свойство	Описание
property Count: Integer;	Количество элементов списка. Это свойство изменяется при добавлении или удалении элемента
property Items (Index: Integer): Pointer;	Возвращает указатель на элемент списка по его индексу. Самый первый элемент списка имеет индекс 0
property List: pPointerList;	Возвращает указатель на массив элементов списка

Тип pPointerList определен следующим образом:

type

```
pPointerList = ^TPointerList;
TPointerList = array [0..MaxListSize] of Pointer;
```

Константа MaxListSize в Delphi 1 ограничена значением 16 379 элементов. Для следующих версий Delphi она ограничивается доступной памятью.

Следует учесть, что свойство Count определяет количество элементов списка, в то время как Capacity — текущую емкость списка. Если при добавлении очередного элемента обнаруживается, что емкость списка исчерпана, происходит наращивание емкости на фиксированную величину (для Count < 5 — на 4 элемента, для 4 < Count < 8 — на 8, для Count > 7 — на 16). При этом сначала резервируется память для размещения расширенного массива указателей, затем в нее копируется содержимое старого массива, после чего старый массив указателей уничтожается (занимаемая им память возвращается Windows).

СОВЕТ Если вы заранее знаете, сколько элементов необходимо поместить в список, установите в начале работы нужное значение в свойство Capacity — это снизит непроизводительные затраты времени на расширение списка.

В табл. 13.2 перечислены некоторые методы класса TList.

Таблица 13.2. Методы класса TList

Метод	Описание
function Add(Item: Pointer): Integer;	Добавляет элемент Item в конец списка и возвращает его индекс
procedure Clear;	Очищает список, удаляя из него все элементы. Не освобождает память, связанную с каждым удаленным элементом. Устанавливает в свойства Count и Capacity значение 0
procedure Delete(Index: Integer);	Удаляет из списка элемент с индексом Index. Все элементы, расположенные за удаляемым, смещаются на одну позицию вверх
procedure Exchange(Index1, Index2: Integer);	Меняет местами элементы с индексами Index1 и Index2
function Expand: TList;	Расширяет массив элементов, увеличивая Capacity

Метод	Описание
function First: Pointer;	Возвращает указатель на самый первый элемент списка
function IndexOf (Item: Pointer): Integer;	Отыскивает в списке элемент Item и возвращает его индекс
procedure Insert (Index: Integer; Item: Pointer);	Вставляет элемент Item в позицию Index списка: новый элемент получает индекс Index, все элементы с индексами Index и больше увеличивают свой индекс на 1. При необходимости расширяет список
function Last: Pointer;	Возвращает указатель на последний элемент списка
procedure Move (CurIndex, NewIndex: Integer);	Перемещает элемент в списке с позиции CurIndex в позицию NewIndex. Все элементы старого списка с индексами от CurIndex - 1 до NewIndex уменьшают свой индекс на 1
procedure Pack;	Упаковывает список: удаляет пустые элементы в конце массива индексов
function Remove (Item: Pointer): Integer;	Отыскивает в списке элемент Item и удаляет его
procedure Sort (Compare: TListSortCompare);	Сортирует коллекцию с помощью функции Compare

Методы Add и Insert получают указатель на вставляемый элемент. Чтобы воспользоваться ими, программист должен сам разместить в куче данные и получить соответствующий указатель. Точно так же методы Delete, Remove и Clear не уничтожают распределенные в памяти данные, которые при необходимости программист должен уничтожить сам. Например:

```

var
  List: TList;
  Item: Pointer;
  Value: AnyType;
begin
  List := TList.Create; // Создаем список
  Item := New(Value);   // Размещаем в куче данные
  List.Add(Item);       // Добавляем элемент к списку
  ...
  List.Remove(Item);    // Удаляем элемент из списка
  Dispose(Item);        // Удаляем его из кучи
  ...
  List.Free;            // Удаляем ненужный список
end;
```

Метод Sort сортирует список по критерию, устанавливаемому функцией Compare. Тип TListSortCompare определен следующим образом:

```

TListSortCompare = function (Item1, Item2: Pointer):
  Integer;
```

Таким образом, функция Compare получает указатели на два элемента списка. Результат сравнения:

- любое отрицательное число, если $\text{Item1}^{\wedge} < \text{Item2}^{\wedge}$;
- 0, если $\text{Item1}^{\wedge} = \text{Item2}^{\wedge}$;
- любое положительное число, если $\text{Item1}^{\wedge} > \text{Item2}^{\wedge}$.

Критерий сравнения данных устанавливается программистом и реализуется в функции Compare.

В следующем примере (проект Source\Chap_13\TList\List.dpr) в список List помещается 20 случайных вещественных чисел, равномерно распределенных в диапазоне 0...1. Список сортируется по возрастанию чисел и отображается в компоненте mmOutput (многострочное текстовое поле из учебной формы fmExample).

type

PDouble = ^Double;

function Comp(Item1, Item2: Pointer): Integer;

// С помощью этой функции реализуется сортировка чисел

begin

if PDouble(Item1)[^] < PDouble(Item2)[^] **then**

Result := -1

else if PDouble(Item1)[^] > PDouble(Item2)[^] **then**

Result := 1

else

Result := 0

end;

procedure TfmExample.bbRunClick(Sender: TObject);

// Обработчик нажатий кнопки bbRun

// выполняет основную работу

var

k: Integer;

List: TList;

pD: PDouble;

begin

List := TList.Create; *// Создаем список*

for k := 1 **to** 20 **do** *// Наполняем его*

begin

New(pD); *// Резервируем память*

pD[^] := Random; *// Помещаем в нее случайное число*

List.Add(pD); *// Добавляем к списку*

end;

List.Sort(Comp); *// Сортируем список по возрастанию*

mmOutput.Lines.Clear; *{ Очищаем поле mmOutput.}*

В следующем цикле наполняем *mmOutput* и уничтожаем элементы *List* }

```
for k := 0 to List.Count-1 do
begin
    pD := List[k];           // Очередное число из списка
    mmOutput.Lines.Add(FloatToStr(pD^)); {Помещаем
                                   в mmOutput}
    Dispose(pD)              // Уничтожаем число
end;
List.Free;                  // Уничтожаем список
end;
```

Классы TStrings и TStringList — наборы строк и объектов

Класс TStrings

Абстрактный класс TStrings инкапсулирует поля и методы для работы с наборами строк. От него порождены многочисленные специализированные потомки, обслуживающие наборы строк в таких компонентах, как TComboBox, TListBox, TRichEdit и др. Эти классы (TComboBoxStrings, TListBoxStrings, TRichEditStrings и др.) объявляются в разделах **Implementation** соответствующих модулей (StdCtrls, ComCtrls и др.) и поэтому скрыты от браузера Delphi и не включены в справочную службу. Единственным доступным наследником TStrings является TStringList — полнофункциональный класс общего назначения.

Замечательной особенностью класса TStrings и его потомков является то обстоятельство, что элементами наборов служат пары *строка-объект*, в которых *строка* — собственно строка символов, а *объект* — объект любого класса Delphi. Такая двойственность позволяет сохранять в TStrings объекты с текстовыми примечаниями, сортировать объекты, отыскивать нужный объект по его описанию и т. д. Кроме того, в качестве объекта может использоваться потомок TStrings, что позволяет создавать многомерные наборы строк.

Набор строк технически реализуется подобно TList — в виде массива указателей. Свойство Capacity показывает текущую длину этого массива, а свойство Count — количество элементов, занятых в нем. Если при добавлении очередного элемента Capacity окажется меньше Count, происходит автоматическое расширение массива. При этом в динамической памяти резервируется место для размещения Capacity + 16 указателей, в новый массив переписывается содержимое старого массива, после чего старый массив уничтожается. Если вам известно количество элементов в создаваемом наборе строк, имеет смысл заранее нужным образом установить свойство Capacity, чтобы сократить непроизводительные расходы на многократные расширения массива указателей.

Свойство CommaText интерпретирует содержимое набора строк в виде одной длинной строки с элементами вида "Первая строка", "Вторая строка", "Третья строка" и т. д. (каждая строка набора заключается в двойные кавычки и отде-

ляется от соседней строки запятой; если в строке встречается символ двойной кавычки, он удваивается). Свойство `Text` интерпретирует содержимое набора в виде одной длинной строки с элементами, разделенными стандартным признаком `EOLN (#13#10)`.

Свойства `Names` и `Values` обрабатывают строки вида `Name=Value`. Такие строки широко используются в различных файлах инициализации, например, в файле `WIN.INI`.

Методы `Add`, `Append`, `Insert`, `Clear` и т. п. в классе `TStrings` абстрактные. Связано это с тем, что класс инкапсулирует их и таким образом делает доступными во всех потомках, но он при этом не накладывает никаких ограничений на то, как располагаются в памяти строки и объекты. Каждый потомок решает эту задачу наиболее удобным для него способом. Например, потомок `TStringList` располагает строки и объекты в общей динамической памяти, для чего перекрывает все абстрактные методы своего родителя. Замечу, что если вы создадите экземпляр класса `TStrings` с помощью его конструктора `Create`, компилятор предупредит вас о том, что этот экземпляр содержит абстрактные методы, так что пользоваться им нужно лишь в исключительных случаях.

Класс `TStringList`

`TStringList` представляет собой полнофункциональный класс общего назначения и является прямым потомком `TStrings`. Помимо перекрытых абстрактных методов своего родителя, класс включает в себя дополнительные методы и свойства, перечисленные в табл. 13.3.

Таблица 13.3. Некоторые свойства и методы класса `TStringList`

Метод или свойство	Описание
property <code>Duplicates:</code> <code>TDuplicates;</code>	Свойство, позволяющее управлять возможностью размещения в наборе двух и более идентичных строк
property <code>Sorted: Boolean;</code>	Признак необходимости сортировки строк в алфавитном порядке
property <code>OnChange:</code> <code>TNotifyEvent;</code>	Определяет реакцию на изменение набора строк. Возникает после последнего изменения
property <code>OnChanging:</code> <code>TNotifyEvent;</code>	Определяет реакцию на изменение набора строк. Возникает до очередного изменения
function <code>Find(const S: String;</code> var <code>Index: Integer): Boolean;</code>	Ищет в наборе строку <code>S</code> и в случае успеха в параметре <code>Index</code> возвращает ее индекс

При `Sorted = True` строки набора автоматически сортируются в алфавитном порядке. При этом свойство `Duplicates` разрешает коллизию, связанную с добавлением в набор строки, идентичной одной из ранее вставленных. Если `Duplicates = duIgnore`, идентичная строка отвергается и программе ничего об этом не сообщается; если `Duplicates = duError`, возбуждается исключение `EListError`; значение `Duplicates = duAccept` разрешает вставлять в набор сколько угодно идентичных строк.

Следует заметить, что сортировка строк в 32-разрядных версиях Windows осуществляется не совсем так, как это происходит в MS-DOS или Windows 3.x. Чтобы убедиться в этом, советую запустить следующий вариант учебной программы (проект Source\Chap_13\TStringList\List.dpr, рис. 13.1). В ней обработчик события OnClick кнопки bbRun создает два списка строк — List1 и List2. Список List1 первоначально способен сортировать строки (в его свойство Sorted устанавливается значение True). Затем в цикле от 32 до 255 (0...31 — это коды служебных символов) происходит наполнение обоих списков: в List1 помещается строка, содержащая символ и его код, а в List2 — наоборот, сначала код, а затем сам символ. Поскольку символьное представление кода дополняется до трех символов (в двузначных числах слева добавляется ведущий ноль), строки в списке List1 окажутся отсортированными по символу, а в списке List2 — по коду символа. Затем оба списка для наглядности объединяются и помещаются в многострочное поле mmOutput.

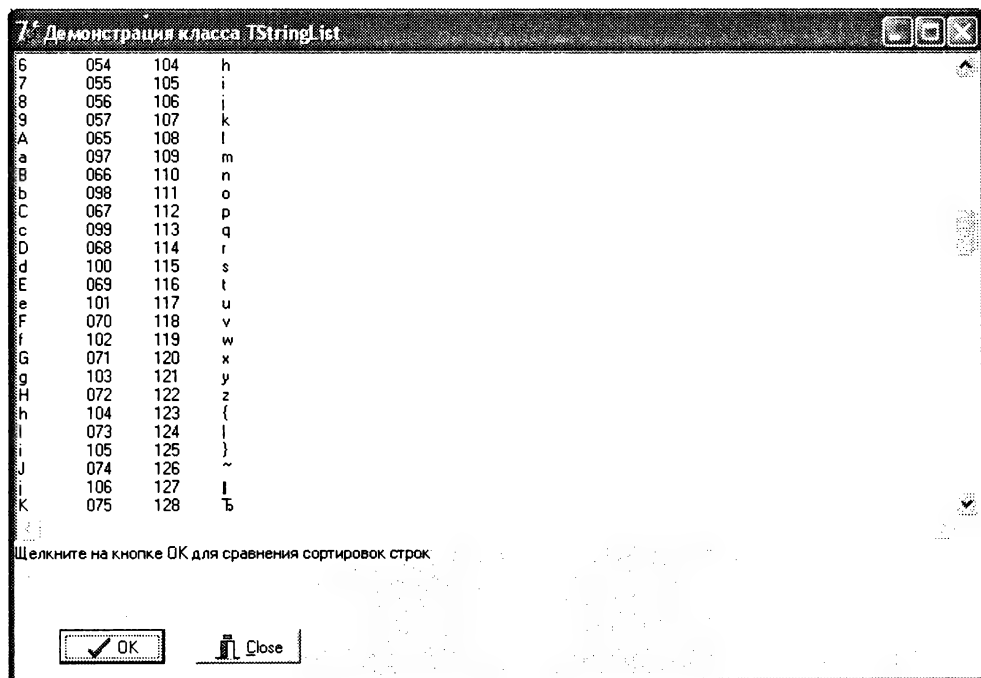


Рис. 13.1. Сравнение методов сортировки строк для разных версий Windows

```

procedure TfmExample.bbRunClick(Sender: TObject);
var
  k: Byte;
  List1, List2: TStringList;
  S: String;
begin
  // Создаем два списка
  List1 := TStringList.Create;

```

```

List1.Sorted := True;
List2 := TStringList.Create;
// Цикл наполнения списков монотонно возрастающими кодами
for k := 32 to 255 do
begin
    S := IntToStr(k); // Код символа
    if k < 100 then
        S := '0' + S; // Двузначное число дополняем ведущим
                        // нулем
// Формируем строку List1 из символа + табуляция + код
// символа
    List1.Add(Char(k) + #9 + S);
    // Формируем строку List2 из кода + табуляция + символ
    List2.Add(S + #9 + Char(k));
end;
// Объединяем оба списка
List1.Sorted := False; // Отключаем сортировку List1
for k := 0 to List1.Count-1 do
    List1[k] := List1[k] + #9 + List2[k];
// Переносим результат в mmOutput
mmOutput.Lines.Assign(List1);
// Уничтожаем List1 и List2
List1.Free;
List2.Free;
end;

```

Следует прокомментировать два момента. Во-первых, все строки List1 помещаются в свойство Lines многострочного поля mmOutput с помощью единственного оператора:

```
mmOutput.Lines.Assign(List1);
```

Это возможно из-за того, что mmOutput.Lines и List1 имеют общего родителя TStringList, который умеет копировать родственные наборы строк с помощью своего метода Assign. Во-вторых, перед объединением строк мы отключили свойство Sorted списка List1. Если этого не сделать, любое изменение строк в отсортированном списке привело бы к исключению EStringListError.

Как видим, строки в 32-разрядных версиях Windows сортируются не с учетом внутреннего кода символа, а с учетом «смысла»: в начале располагаются все знаки препинания и разного рода «мусор», затем идут цифры, буквы латинского алфавита и символы кириллицы. Буквы упорядочены парами — сначала прописная, за ней строчная — и не идут сплошным массивом. «Виновником» такой сортировки является API-функция AnsiCompareText, к которой обращается метод TStringList.QuickSort. Если вам понадобится отсортировать так, как это принято в MS-DOS или Windows 3.x, то есть по коду символа, сделайте следующее.

1. Разыщите в каталоге Source\Rtl\Common исходный файл CLASSES.PAS и скопируйте его в ваш рабочий каталог. Если вы не найдете этого файла, вы не сможете изменить способ сортировки, так как в классе TStringList метод QuickSort, отвечающий за сортировку, определен в секции **private** и, следовательно, недоступен наследникам.

2. В раздел **Implementation** этого модуля сразу после предложения **Uses** поместите такое описание функции `AnsiCompareText`:

```
function AnsiCompareText(S1, S2: String): Integer;
begin
  if S1 < S2 then Result := -1
  else if S1 = S2 then Result := 0
  else Result := 1
end;
```

3. С помощью команды **Project ► Add To Project** добавьте измененный модуль `Classes` к вашему проекту и сделайте новый прогон программы. После этого оба столбца будут идентичны.

Графический инструментарий

Богатство изобразительных возможностей Windows связано с так называемым дескриптором контекста графического устройства (`Device Context, DC`) и тремя входящими в него инструментами — шрифтом, пером и кистью. В Delphi созданы специализированные классы-надстройки, существенно упрощающие использование графических инструментов Windows: для канвы — класс `TCanvas`, для шрифта — `TFont`, для пера — `TPen` и для кисти — `TBrush`. Связанные с этими классами объекты автоматически создаются для всех видимых элементов и становятся доступны программе через свойства `Canvas`, `Font`, `Pen` и `Brush`.

Класс TFont

С помощью класса `TFont` создается объект-шрифт для любого графического устройства (экрана, принтера, плоттера и т. п.).

Наиболее важные свойства класса перечислены в табл. 13.4.

Таблица 13.4. Свойства класса `TFont`

Свойство	Описание
property Charset: <code>TFontCharset</code> ;	Набор символов. Для русскоязычных программ это свойство обычно имеет значение <code>DEFAULT_CHARSET</code> или <code>RUSSIAN_CHARSET</code> . Используйте значение <code>OEM_CHARSET</code> для отображения текста MS-DOS (альтернативная кодировка)
property Color: <code>TColor</code> ;	Цвет шрифта
property Height: <code>Integer</code> ;	Высота шрифта в пикселах экрана
property Name: <code>TFontName</code> ;	Имя шрифта. По умолчанию имеет значение <code>MS Sans Serif</code>
property Size: <code>Integer</code> ;	Высота шрифта в пунктах (1/72 дюйма). Изменение этого свойства автоматически изменяет свойство <code>Height</code> и наоборот
property Style: <code>TFontStyles</code> ;	Стиль шрифта. Может принимать значение как комбинацию следующих признаков: <code>fsBold</code> (жирный), <code>fsItalic</code> (курсив), <code>fsUnderline</code> (подчеркнутый), <code>fsStrikeOut</code> (перечеркнутый)

Для некоторых случаев может оказаться полезным метод:

```
procedure Assign(Source: TPersistent);
```

С помощью этого метода значения свойств шрифтового объекта Source присваиваются свойствам текущего шрифта. Метод можно использовать для создания шрифта принтера по экранному шрифту и наоборот.

В проекте Source\Chap_13\TFont\FontDemo.dpr иллюстрируются основные приемы работы со шрифтом. Для его повторения удалите из учебной программы поле mmOutput, поместите на его место компонент PaintBox (вкладка System), назовите его pbOut и поместите в его свойство Align значение alClient. На нижней панели разместите еще две кнопки (одна для выбора шрифта, другая для вывода надписи), назовите их bbFont и bbOut, а в их свойства Caption поместите соответственно значения Шрифт и Вывод (рис. 13.2).

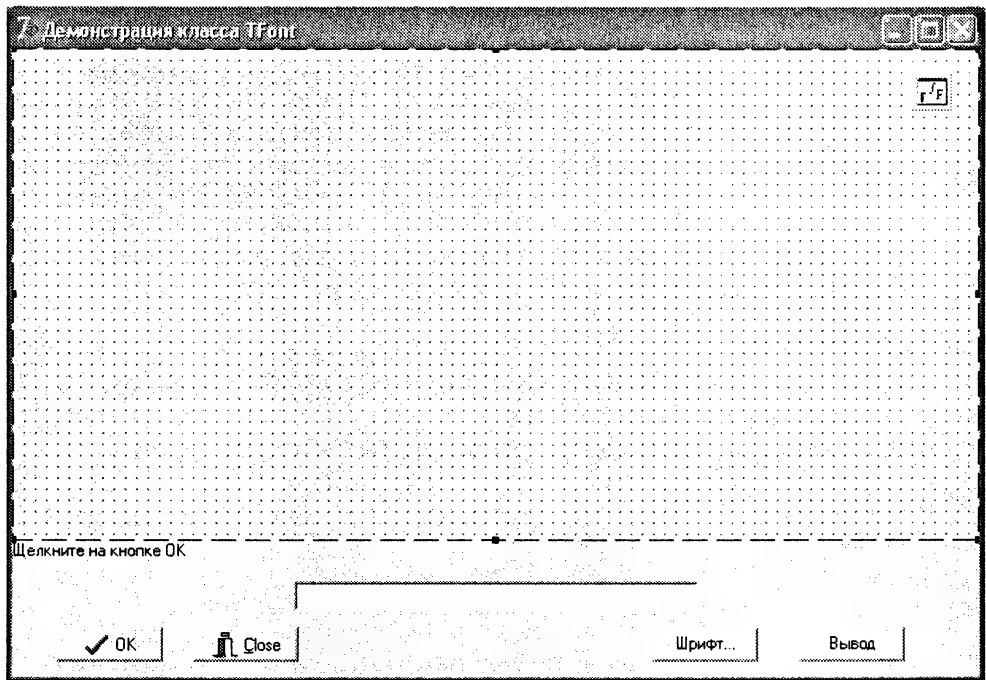


Рис. 13.2. Окно программы на этапе конструирования

На любое место формы поместите стандартное диалоговое окно FontDialog (вкладка Dialogs) и напишите такой обработчик события OnClick для кнопки bbFont:

```
procedure TfmExample.bbFontClick(Sender: TObject);
begin
    if FontDialog1.Execute then
        pbOut.Canvas.Font := FontDialog1.Font
end;
```

Спрячьте компоненты `edInput`, `bbFont`, `bbOut` (поместите в их свойства `Visible` значения `False`). Обработчики `OnClick` кнопок `bbRun` и `bbOut` должны быть такими:

```
procedure TfmExample.bbRunClick(Sender: TObject);
{Выводит исходный текст, прячет кнопку bbRun, показывает
поле edInput и кнопки bbFont и bbOut}
begin
  with pbOut.Canvas, Font do
    begin
      TextOut(10, 10, 'MS Sans Serif, Size=10');
      Size := 20;
      TextOut(10, 30, 'MS Sans Serif, Size=20');
      Name := 'Courier';
      Style := [fsBold];
      TextOut(10, 70, 'Courier, Syze=20');
      Name := 'Times New Roman';
      Style := [fsItalic, fsUnderline, fsBold];
      TextOut(10, 110, 'Times Nes Roman, курсив, жирный, '+'
        'подчерк. ');
    end;
    edInput.Show;
    bbFont.Show;
    bbOut.Show;
    bbRun.Hide;
    lbOutput.Caption := 'Выберите шрифт, введите текст и '+'
      'щелкните на кнопке "Вывод"!';
end;

procedure TfmExample.bbOutClick(Sender: TObject);
{Выводит текст из поля edInput текущим шрифтом}
begin
  if edInput.Text <> '' then
    pbOut.Canvas.TextOut(10, 150, edInput.Text);
    edInput.Text := ''; // Очищаем поле ввода
    edInput.SetFocus    // Передаем ему фокус ввода
end;
```

Вид окна работающей программы показан на рис. 13.3.

Некоторые пояснения к программе. Мы воспользовались шрифтом канвы компонента `PaintBox1` (канва обсуждается ниже в подразделе «Класс `TCanvas`»). Для изменения стиля шрифта его свойству `Style` следует присвоить произвольный набор флагов `fsXXXX` — значение типа `TFontStyle`. Этот тип определен как множество из элементов перечисленного типа. Поэтому в правой части оператора присваивания должен быть конструктор множества, даже если устанавливается лишь один флаг:

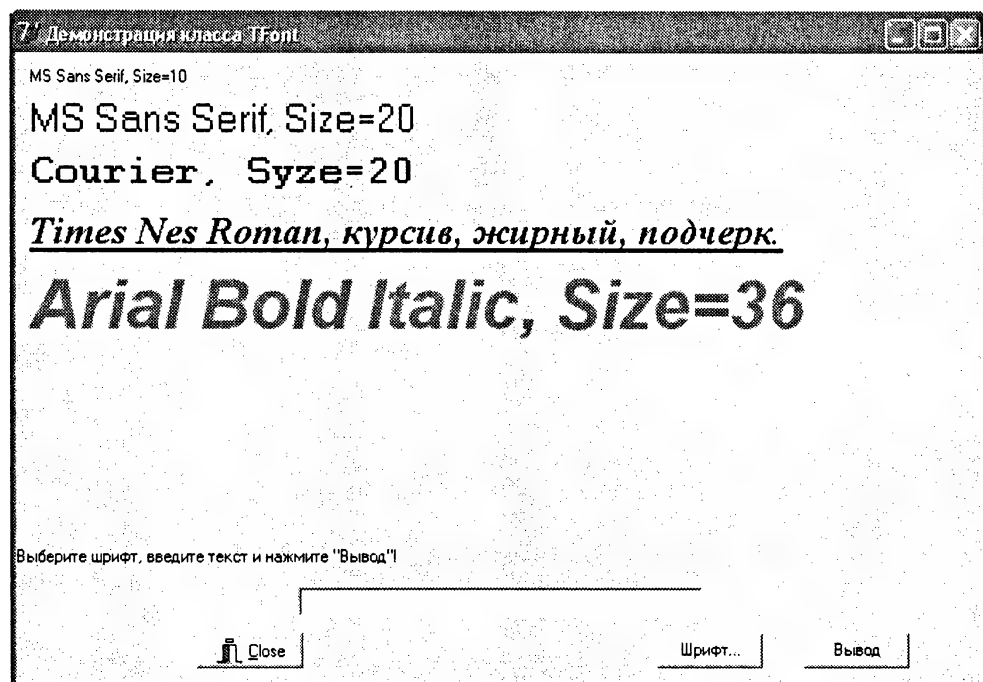


Рис. 13.3. Вид окна работающей программы

```
Style := [fsBold];
```

Вывод текста реализуется методом `TextOut` канвы, которому передаются координаты левого верхнего угла надписи и текст.

Класс `TPen`

С помощью класса `TPen` создается объект-перо, служащий для вычерчивания линий.

В табл. 13.5 перечислены свойства класса.

Таблица 13.5. Свойства класса `TPen`

Свойство	Описание
property Color: <code>TColor;</code>	Цвет вычерчиваемых пером линий
property Mode: <code>TPenMode;</code>	Определяет способ взаимодействия линий с фоном (см. ниже)
property Style: <code>TPenStyle;</code>	Определяет стиль линий (см. ниже). Учитывается только для толщины линий 1 пиксел. Для толстых линий стиль всегда <code>psSolid</code> (сплошная)
property Width: <code>Integer;</code>	Толщина линий в пикселах экрана

Свойство Mode может принимать одно из следующих значений:

- pmBlack — линии всегда черные, свойства Color и Style игнорируются;
- pmWhite — линии всегда белые, свойства Color и Style игнорируются;
- pmNop — цвет фона не меняется (линии не видны);
- pmNot — инверсия цвета фона, свойства Color и Style игнорируются;
- pmCopy — цвет линий определяется свойством Color пера;
- pmNotCopy — инверсия цвета пера, свойство Style игнорируется;
- pmMergePenNot — комбинация цвета пера и инверсионного цвета фона;
- pmMaskPenNot — комбинация общих цветов для пера и инверсионного цвета фона, свойство Style игнорируется;
- pmMergeNotPen — комбинация инверсионного цвета пера и фона;
- pmMaskNotPen — комбинация общих цветов для инверсионного цвета пера и фона, свойство Style игнорируется;
- pmMerge — комбинация цветов пера и фона;
- pmNotMerge — инверсия цветов пера и фона, свойство Style игнорируется;
- pmMask — общие цвета пера и фона;
- pmNotMask — инверсия общих цветов пера и фона;
- pmXor — объединение цветов пера и фона операцией XOR;
- pmNotXor — инверсия объединения цветов пера и фона операцией XOR.

Стили линий показаны в верхней части рис. 13.4. Этот рисунок создан с помощью программы Source\Chap_13\TPen\PenDemo.dpr.

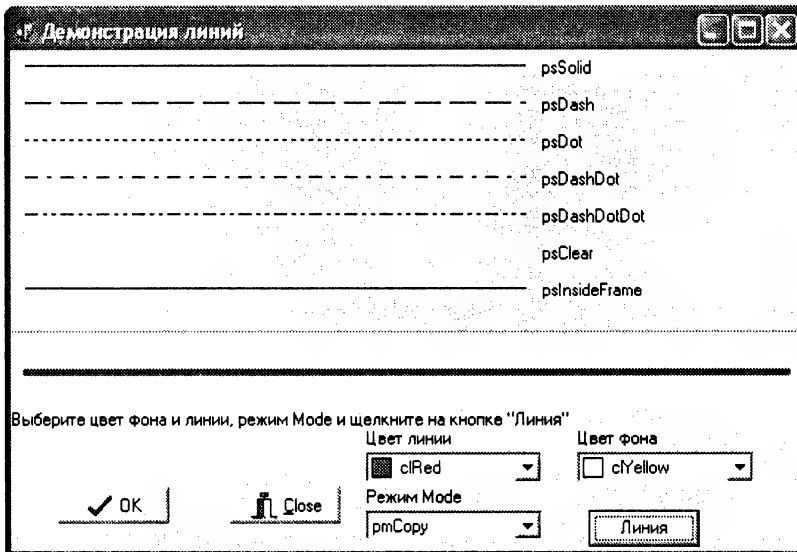


Рис. 13.4. Стили линий

Для повторения примера удалите из формы учебной программы поле mmOutput и на его место поместите два компонента PaintBox (вкладка System): один по щелчку на кнопке bbRun покажет все стили линий, другой продемонстрирует взаимодействие цветов линии и фона для различных значений свойства Mode. Назовите один из них pbPen, другой — pbOut. Для первого в свойство Align поместите значение alBottom, для второго — alClient. Удалите из формы поле edInput и разместите на нижней панели два компонента ColorBox (вкладка Additional). Назовите один cbPen (с его помощью пользователь будет выбирать цвет пера), а другой — cbBack (выбор цвета фона). На нижней же панели разместите список cbMode (с его помощью выбирается режим Mode взаимодействия пикселей линии с пикселями фона) и кнопку bbPen (вычерчивает линию толщиной 5 пикселей с выбранным режимом Mode). В листинге 13.1 показан текст модуля формы.

Листинг 13.1. Текст модуля формы примера PenDemo.dpr¹

```

procedure TfmExample.bbRunClick(Sender: TObject);
var
    k: Integer;
const
    Names: array [0..6] of String[30]= ('psSolid', 'psDash',
        'psDot', 'psDashDot', 'psDashDotDot', 'psClear',
        'psInsideFrame');
begin
    // Вычерчиваем 7 линий разного стиля:
    with pbOut.Canvas, Pen do
        begin
            for k := 0 to 6 do
                begin
                    Style := TPenStyle(k);
                    // Устанавливаем очередной стиль
                    MoveTo(10, k*25+10); // Смещаем перо к началу линии
                    LineTo(350, k*25+10); // Вычерчиваем линию
                    TextOut(360, k*25+5, Names[k])
                    // Выводим название стиля
                end
            end;
            lbOutput.Caption := 'Выберите цвет фона и линии, режим
                Mode '+'и щелкните на кнопке "Линия"';
        end;

    const // 16 названий возможных значений свойства Mode
    PenMode: array [0..15] of String[13]=('pmBlack',
        'pmWhite', 'pmNop', 'pmNot', 'pmCopy', 'pmNotCopy',
        'pmMergePenNot', 'pmMaskPenNot', 'pmMergeNotPen',

```

¹ Все представленные в книге тексты программ можно загрузить с сайта издательства по адресу <http://www.piter.com/download> — Примеч. ред.

```
'pmMaskNotPen', 'pmMerge', 'pmNotMrege', 'pmMask',  
'pmNotMask', 'pmXOR', 'pmNotWor');
```

```
procedure TfmExample.bbPenClick(Sender: TObject);  
var  
    X, Y, L: Integer;  
begin  
    with pbPen.Canvas do  
        begin  
            Pen.Width := 5;  
            // Закрашиваем фон:  
            Brush.Color := cbBack.Colors[cbBack.ItemIndex];  
            FillRect(ClipRect);  
            // Устанавливаем цвет линии и режим:  
            Pen.Color := cbPen.Colors[cbPen.ItemIndex];  
            Pen.Mode := TPenMode(cbMode.ItemIndex);  
            // Центрируем линию в поле pbPen и выводим ее:  
            X := 10;  
            Y := ClipRect.Bottom div 2;  
            L := ClipRect.Right-10;  
            MoveTo(X, Y);  
            LineTo(L, Y)  
        end  
    end;
```

```
procedure TfmExample.FormCreate(Sender: TObject);  
// Наполняет список cbMode в момент создания формы  
var  
    k: Integer;  
begin  
    cbMode.Items.Clear;  
    for k := 0 to 15 do  
        cbMode.Items.Add(PenMode[k])  
    end;
```

Обратите внимание на оператор:

```
Style := TPenStyle(k); //Устанавливаем очередной стиль
```

Значениями свойства TPenStyle могут быть элементы перечисленного стиля, которые есть не что иное, как целочисленные значения. В программе можно написать:

```
Style := psNot;
```

В этом случае компилятор по описанию типа TPenStyle найдет нужную целочисленную константу и подставит ее в выражение. Следующий оператор будет ошибочным, так как переменной Style перечисленного типа TPenStyle присваивается строковое значение:

```
Style := 'psNot';
```

Мы воспользовались тем, что порядок перечисления значений в константе Names соответствует перечислению значений типа TPenStyle и применили автоматически определенную функцию приведения типов TPenStyle для преобразования целочисленного значения в значение перечисленного типа (функции приведения автоматически создаются для каждого типа; их имена совпадают с именами соответствующих типов). Заметьте, что следующий оператор недопустим, поскольку свойству перечисленного типа присваивается целочисленное значение:

```
Style := k;
```

Класс TBrush

Объекты класса TBrush (кисти) служат для заполнения внутреннего пространства замкнутых фигур.

В табл. 13.6 представлены свойства класса.

Таблица 13.6. Свойства класса TBrush

Свойство	Описание
property Bitmap: TBitmap;	Содержит растровое изображение, которое будет использоваться кистью. Если это свойство определено, свойства Color и Style игнорируются
property Color: TColor;	Цвет кисти
property Style: TBrushStyle;	Стиль кисти (см. ниже)

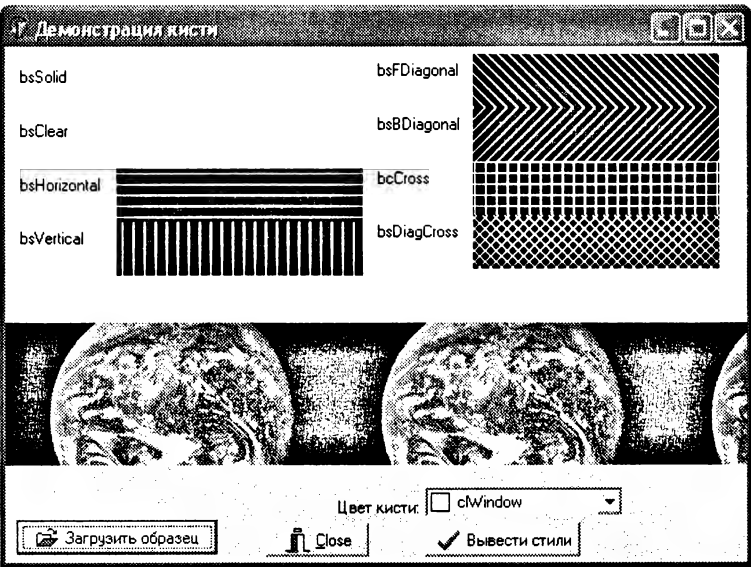


Рис. 13.5. Стили кисти

Замечу, что свойство `Bitmap` содержит ссылку на *объект* класса `TBitmap`. Перед обращением к свойству этот объект нужно создать. Свойство `Color` определяет цвет рисунка (узора) кисти, но не цвет фона: цвет фона выбирается автоматически так, чтобы линии узора контрастировали с фоном. В программе `Source\Chap_13\TBrush\BrushDemo.dpr` иллюстрируются стандартные стили кисти (верхняя часть рис. 13.5) и использование свойства `Bitmap` (нижняя часть).

Я не буду утомлять вас подробным описанием процедуры создания соответствующей формы: вы ее найдете в упомянутом проекте, а при желании воспроизвести пример самостоятельно руководствуйтесь рис. 13.5 и приводимым ниже листингом 13.2 главного модуля программы.

Листинг 13.2. Листинг модуля `BrushU.pas`

```
unit BrushU;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Buttons, ExtCtrls,
  ExtDlgs;

type
  TfmExample = class(TForm)
    Panell: TPanel;
    bbRun: TBitBtn;
    bbClose: TBitBtn;
    pbOut: TPaintBox;
    pbBit: TPaintBox;
    Splitter1: TSplitter;
    cbBrushColor: TColorBox;
    BitBtn1: TBitBtn;
    OPDlg: TOpenPictureDialog;
    Labell: TLabel;
    procedure bbRunClick(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  fmExample: TfmExample;
```

implementation

```

{$R *.dfm}
procedure TfmExample.bbRunClick(Sender: TObject);
// Воспроизводит стили кисти с цветом, заданным списком
// cbBrushCilor
const
  Names: array [0..7] of String[12] = ('bsSolid','bsClear',
    'bsHorizontal','bsVertical','bsFDiagonal',
    'bsBDiagonal','bcCross','bsDiagCross');
var
  k, X, Y, H, W: Integer;
  R: TRect;
begin
  X := 10; // Текущая левая координата
  Y := 10; // Текущая верхняя координата области прорисовки
  with pbOut.Canvas do
    begin
      // Вначале очищаем весь рисунок:
      Brush.Color := clWhite;
      Brush.Style := bsSolid;
      FillRect(ClipRect);
      // Вычисляем ширину (W) и высоту (H) изображения:
      W := ClipRect.Right div 2-90;
      H := ClipRect.Bottom div 4-10;
      // Цикл вывода 8 стилей:
      for k := 0 to 7 do
        begin
          if k=4 then
            begin // Переход к правой колонке
              X := W+90;
              Y := 5
            end;
          Brush.Style := TBrushStyle(k);
          if k<>1 then
            Brush.Color :=
              cbBrushColor.Colors[cbBrushColor.ItemIndex];
          R.Top := Y;
          inc(Y, H);
          R.Bottom := R.Top+H;
          R.Left := X+70;
          R.Right := R.Left+W;
          FillRect(R);
        end
      end
    end

```

```

    // Выводим название стиля:
    Brush.Color := clWhite;
    TextOut(X, R.Top+5, Names[k]);
    R.Top := R.Top+N;
end
end
end;

procedure TfmExample.BitBtn1Click(Sender: TObject);
// Выбираем фоновый узор
begin
    with pbBit.Canvas, Brush do
    begin
        Bitmap := TBitmap.Create; // Создаем объект-изображение
        try
            if OPDlg.Execute then
                Bitmap.LoadFromFile(OPDlg.FileName); // Загружаем его
                FillRect(ClipRect); // Закрашиваем всю область
            finally
                Bitmap.Free // Освобождаем объект
            end
        end
    end
end;

```

Обратите внимание: при назначении кисти нового стиля автоматически очищается свойство `Color`, которое следует установить заново, если только стиль — не `bsClear`. Перед загрузкой объекта `Bitmap` его нужно создать, а после использования — уничтожить. Поскольку прорисовка слишком крупных изображений может вызвать исключение, этот фрагмент текста защищен блоком **try...finally**. Перед выводом текста прямоугольник вывода заливается цветом кисти. Обратите внимание на оператор перед обращением к методу `TextOut` канвы:

```
Brush.Color := clWhite;
```

Если его убрать, текст будет выведен на фоне, определяемым выбором в списке `cbBrushColor`.

Класс TCanvas

Этот класс создает «канву», на которой можно рисовать чертежными инструментами — пером, кистью и шрифтом. Объекты класса `TCanvas` автоматически создаются для всех видимых компонентов, которые должны уметь нарисовать себя. Они инкапсулируют объекты `Font`, `Pen`, `Brush`, а также многочисленные методы, использующие эти объекты.

Свойства класса перечислены в табл. 13.7.

Таблица 13.7. Свойства класса TCanvas

Свойство	Описание
property Brush: TBrush;	Объект-кисть
property ClipRect: TRect;	Определяет текущие размеры области, пугжающейся в прорисовке
property CopyMode: TCopyMode;	Устанавливает способ взаимодействия растрового изображения с цветом фона (см. ниже)
property Font: TFont;	Объект-шрифт
property Pen: TPen;	Объект-перо
property PenPos: TPoint;	Определяет текущее положение пера в пикселах относительно левого верхнего угла канвы
property Pixels[X,Y: Integer]: TColor;	Массив пикселей канвы

Свойство CopyMode используется при копировании части одной канвы (источника) в другую (приемник) методом CopyRect и может иметь одно из следующих значений:

- cmBlackness — заполняет область рисования черным цветом;
- cmDestInvert — заполняет область рисования инверсным цветом фона;
- cmMergeCopy — объединяет изображение на канве и копируемое изображение операцией AND;
- cmMergePaint — объединяет изображение на канве и копируемое изображение операцией OR;
- cmNotSrcCopy — копирует на канву инверсное изображение источника;
- cmNotSrcErase — объединяет изображение на канве и копируемое изображение операцией OR и инвертирует полученное;
- cmPatCopy — копирует образец источника;
- cmPatInvert — комбинирует образец источника с изображением на канве с помощью операции XOR;
- cmPatPaint — комбинирует изображение источника с его образцом с помощью операции OR, затем полученное объединяется с изображением на канве также с помощью операции OR;
- cmSrcAnd — объединяет изображение источника и канвы с помощью операции AND;
- cmSrcCopy — копирует изображение источника на канву;
- cmSrcErase — инвертирует изображение на канве и объединяет результат с изображением источника операцией AND;
- cmSrcInvert — объединяет изображение на канве и источник операцией XOR;
- cmSrcPaint — объединяет изображение на канве и источник операцией OR;
- cmWhitess — заполняет область рисования белым цветом.

С помощью свойства `Pixels` все пиксели канвы представляются в виде двумерного массива точек с нулевой нижней границей по каждому измерению; точка `Pixels[0, 0]` соответствует левому верхнему пикселу канвы. Изменяя цвет пикселей, можно прорисовывать изображение по отдельным точкам.

Методы класса перечислены в табл. 13.8.

Таблица 13.8. Методы класса `TCanvas`

Метод	Описание
procedure Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);	Чертит дугу эллипса в охватывающем прямоугольнике (X1,Y1) – (X2,Y2). Начало дуги лежит на пересечении эллипса и луча, проведенного из его центра в точку (X3,Y3), а конец — на пересечении с лучом из центра в точку (X4,Y4). Дуга чертится против часовой стрелки (см. ниже)
procedure Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);	Чертит сегмент эллипса в охватывающем прямоугольнике (X1,Y1) – (X2,Y2). Начало дуги сегмента лежит на пересечении эллипса и луча, проведенного из его центра в точку (X3,Y3), а конец — на пересечении с лучом из центра в точку (X4,Y4). Дуга сегмента чертится против часовой стрелки, а начальная и конечная точки дуги соединяются прямой (см. ниже)
procedure CopyRect (Dest: TRect; Canvas: TCanvas; Source: TRect);	Копирует изображение Source канвы Canvas в участок Dest текущей канвы
procedure Draw(X, Y: Integer; Graphic: TGraphic);	Осуществляет прорисовку графического объекта Graphic так, чтобы левый верхний угол объекта расположился в точке (X,Y)
procedure DrawFocusRect (const Rect: TRect);	Прорисовывает прямоугольник с помощью операции XOR, поэтому повторная прорисовка уничтожает ранее вычерченный прямоугольник. Используется, в основном, для прорисовки нестандартных интерфейсных элементов при получении ими фокуса ввода и при потере его
procedure Ellipse (X1, Y1, X2, Y2: Integer);	Чертит эллипс в охватывающем прямоугольнике (X1,Y1) – (X2,Y2). Заполняет внутреннее пространство эллипса текущей кистью
procedure FillRect (const Rect: TRect);	Заполняет текущей кистью прямоугольную область Rect, включая ее левую и верхнюю границы, но не затрагивая правую и нижнюю границы
procedure FloodFill (X, Y: Integer; Color: TColor; FillStyle: TFillStyle);	Производит заливку канвы текущей кистью. Заливка начинается с точки (X,Y) и распространяется во все стороны от нее. Если FillStyle=fsSurface, заливка распространяется на все соседние точки с цветом Color. Если FillStyle=fsBorder, наоборот, заливка прекращается на точках с этим цветом
procedure FrameRect (const Rect: TRect);	Очерчивает границы прямоугольника Rect текущей кистью толщиной в 1 пиксел без заполнения внутренней части прямоугольника

продолжение ➤

Таблица 13.8 (продолжение)

Метод	Описание
procedure LineTo (X, Y: Integer);	Чертит линию от текущего положения пера до точки (X,Y)
procedure MoveTo (X, Y: Integer);	Перемещает перо в положение (X,Y) без вычерчивания линий
procedure Pie (X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);	Рисует сектор эллипса в охватывающем прямоугольнике (X1,Y1) – (X2,Y2). Начало дуги лежит на пересечении эллипса и луча, проведенного из его центра в точку (X3,Y3), а конец — на пересечении с лучом из центра в точку (X4,Y4). Дуга чертится против часовой стрелки. Начало и конец дуги соединяются прямыми с ее центром (см. ниже)
procedure Polygon (Points: array of TPoint);	Вычерчивает пером многоугольник по точкам, заданным в массиве Points. Конечная точка соединяется с начальной, и многоугольник заполняется кистью. Для вычерчивания без заполнения используйте метод Polyline
procedure Polyline (Points: array of TPoint);	Вычерчивает пером ломаную прямую по точкам, заданным в массиве Points
procedure Rectangle (X1, Y1, X2, Y2: Integer);	Вычерчивает и заполняет прямоугольник (X1,Y1) – (X2,Y2). Для вычерчивания без заполнения используйте метод FrameRect или Polyline
procedure Refresh;	Устанавливает в канве умалчиваемые шрифт, перо и кисть
procedure RoundRect (X1, Y1, X2, Y2, X3, Y3: Integer);	Вычерчивает и заполняет прямоугольник (X1,Y1) – (X2,Y2) со скругленными углами. Прямоугольник (X1,Y1) – (X3,Y3) определяет дугу эллипса для скругления углов (см. ниже)
procedure StretchDraw (const Rect: TRect; Graphic: TGraphic);	Вычерчивает и при необходимости масштабирует графический объект Graphic так, чтобы он полностью занял прямоугольник Rect
function TextExtent (const Text: String): TSize;	Возвращает ширину и высоту прямоугольника, охватывающего текстовую строку Text
function TextHeight (const Text: String): Integer;	Возвращает высоту прямоугольника, охватывающего текстовую строку Text
procedure TextOut (X, Y: Integer; const Text: String);	Выводит текстовую строку Text так, чтобы левый верхний угол прямоугольника, охватывающего текст, располагался в точке (X,Y)
procedure TextRect (Rect: TRect; X, Y: Integer; const	Выводит текстовую строку Text так, чтобы левый верхний угол прямоугольника, охватывающего текст, располагался в точке (X,Y). Если при этом какая-либо часть надписи

Метод	Описание
<code>Text: String;</code>	выходит из границ прямоугольника Rect, она отсекается и не будет видна
<code>function TextWidth</code> (const Text: String): Integer;	Возвращает ширину прямоугольника, охватывающего текстовую строку Text

На рис. 13.6 показаны параметры обращения к методам Arc, Chord, Pie и RoundRect.

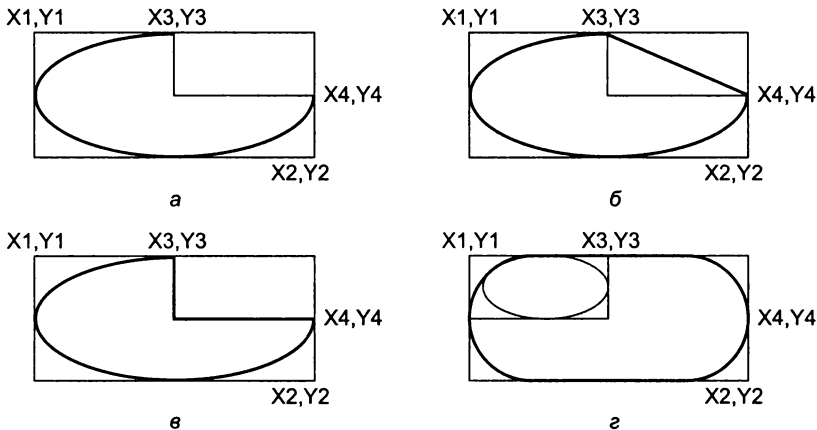


Рис. 13.6. Параметры обращения к методам Arc (а), Chord (б), Pie (в) и RoundRect (г)

Следует отметить два момента. Во-первых, создаваемые на канве изображения безвозвратно теряются при перемещении окна программы, его перекрытии другими окнами, при изменении размеров окна или канвы. Если вам необходимо восстановить изображение, следует вынести его прорисовку в отдельную подпрограмму и вызывать ее в обработчиках событий OnActivate и OnResize. Во-вторых, нельзя рисовать на невидимой канве. Это означает, что свойство Visible компонента, содержащего канву, игнорируется, как только на канве появляется изображение. Это также означает, что нельзя прорисовывать изображение в обработчике OnCreate формы, так как в момент создания окна с канвой оно еще не появилось на экране.

В программе Source\Chap_13\TCanvas\CanvasDemo.dpr (рис. 13.7) иллюстрируются некоторые приемы работы с объектом класса TCanvas.

Классы TGraphic и TPicture

Важное место в графическом инструментарии Delphi занимают классы TGraphic и TPicture.

TGraphic — это абстрактный класс, инкапсулирующий общие свойства и методы трех своих потомков: значка (TIcon), метафайла (TMetafile) и растрового изображения (TBitmap). Общей особенностью потомков TGraphic является то, что обычно они сохраняются в файлах определенного формата. Значки представ-

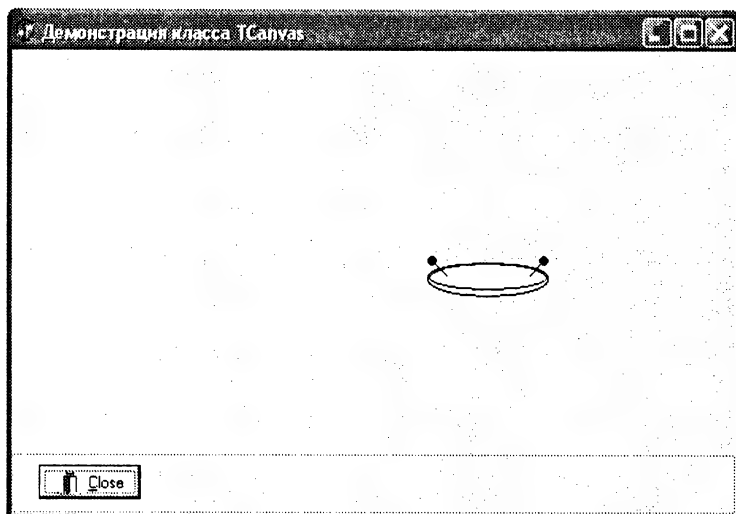


Рис. 13.7. Демонстрационная программа «НЛО»

ляют собой небольшие растровые изображения, снабженные специальными средствами, регулирующими их прозрачность. Для файлов значков обычно используется расширение ICO. Метафайл — это изображение, построенное на графическом устройстве с помощью специальных команд, которые сохраняются в файле с расширением WMF или EMF. Растровые изображения — это произвольные графические изображения в файлах со стандартным расширением BMP.

В табл. 13.9 перечислены свойства класса TGraphic, а в табл. 13.10 — его методы.

Таблица 13.9. Свойства класса TGraphic

Свойство	Описание
property Empty: Boolean;	Содержит True, если с объектом не связано графическое изображение
property Height: Integer;	Содержит высоту изображения в пикселах
property Modified: Boolean;	Содержит True, если графический объект изменялся
property Palette: HPALETTE;	Содержит цветовую палитру графического объекта
property PaletteModified: Boolean;	Содержит True, если менялась цветовая палитра графического объекта
property Transparent: Boolean;	Содержит True, если объект прозрачен для фона, на котором он изображен
property Width: Integer;	Содержит ширину изображения в пикселах

Полнофункциональный класс TPicture инкапсулирует в себе все необходимое для работы с готовыми графическими изображениями — значком, растром или метафайлом. Его свойство Graphic может содержать объект любого из этих типов, обеспечивая нужный полиморфизм методов класса.

В табл. 13.11 и 13.12 перечислены свойства и методы класса TPicture.

Таблица 13.10. Методы класса TGraphic

Метод	Описание
procedure LoadFromClipboardFormat (AFormat: Word; AData: THandle; APalette: HPALETTE);	Ищет в буфере межпрограммного обмена (clipboard) зарегистрированный формат AFormat и, если формат найден, загружает из буфера изображение AData и его палитру APalette
procedure LoadFromFile (const FileName: String);	Загружает изображение из файла FileName
procedure LoadFromStream (Stream: TStream);	Загружает изображение из потока данных Stream
procedure SaveToClipboardFormat (var AFormat: Word; var AData: THandle; var APalette: HPALETTE);	Помещает графическое изображение AData и его цветовую палитру APalette в буфер межпрограммного обмена в формате AFormat
procedure SaveToFile (const FileName: String);	Сохраняет изображение в файле FileName
procedure SaveToStream (Stream: TStream);	Сохраняет изображение в потоке данных Stream

Таблица 13.11. Свойства класса TPicture

Свойство	Описание
property Bitmap: TBitmap;	Интерпретирует графический объект как растровое изображение
property Graphic: TGraphic;	Содержит графический объект
property Height: Integer;	Содержит высоту изображения в пикселах
property Icon: TIcon;	Интерпретирует графический объект как значок
property Metafile: TMetafile;	Интерпретирует графический объект как метафайл
property Width: Integer;	Содержит ширину изображения в пикселах

Таблица 13.12. Методы класса TPicture

Метод	Описание
procedure Assign (Source: TPersistent);	Связывает собственный графический объект Graphic с объектом Source
procedure LoadFromClipboardFormat (AFormat: Word; AData: THandle; APalette: HPALETTE);	Ищет в буфере межпрограммного обмена (clipboard) зарегистрированный формат AFormat и, если формат найден, загружает из буфера изображение AData и его палитру APalette

продолжение ➤

Таблица 13.12 (продолжение)

Метод	Описание
procedure LoadFromFile(const FileName: String);	Загружает изображение из файла FileName
procedure SaveToClipboardFormat (var AFormat: Word; var AData: THandle; var APalette: HPALETTE);	Помещает графическое изображение AData и его цветовую палитру APalette в буфер межпрограммного обмена в формате AFormat
procedure SaveToFile(const FileName: String);	Сохраняет изображение в файле FileName

Глава 14

Общие свойства компонентов

Компонентами в Delphi называются потомки класса `TComponent`. В этой главе описываются общие свойства, методы и события компонентов. В остальных главах книги при описании конкретного компонента эти свойства уже не упоминаются. Следует учесть, что все они относятся, в основном, к классам-родоначальникам — `TObject`, `TPersistent`, `TComponent` и `TControl`. Некоторые из них описаны в секциях **protected** этих классов и могут быть недоступны своим потомкам. Поэтому для получения полного набора методов, свойств и событий конкретного компонента необходимо обращаться к встроенной справочной службе Delphi.

Иерархия компонентов

Все компоненты Delphi порождены от класса `TComponent`, в котором инкапсулированы самые общие свойства и методы компонентов. Предишним `TComponent` является класс `TPersistent`, который произошел непосредственно от базового класса `TObject` (рис. 14.1).

Класс `TPersistent` передает своим потомкам важный виртуальный метод:

procedure Assign(Source: `TPersistent`);

С помощью этого метода поля и свойства объекта `Source` копируются в объект, вызвавший метод `Assign`. Замечу, что обмениваться данными могут все наследники класса `TPersistent` независимо от того, входят ли они в непосредственном родстве по отношению друг к другу или имеют единственного общего предка — класс `TPersistent`.

Класс `TComponent` служит базой для создания как видимых, так и невидимых компонентов. Большая часть видимых компонентов происходит от класса `TControl`. Два наследника этого класса — `TWinControl` и `TGraphicControl` — определяют две группы компонентов: имеющие оконный ресурс (класс `TWinControl` и его потомки) и не имеющие этого ресурса (класс `TGraphicControl` и его потомки). *Оконный ресурс* — это специальный ресурс Windows, предназначенный для

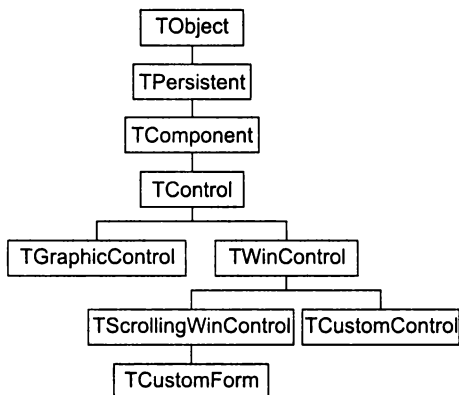


Рис. 14.1. Фрагмент иерархии компонентов Delphi

создания и обслуживания окон. Только оконные компоненты способны получать и обрабатывать сообщения Windows.

Оконный компонент в момент своего создания обращается к Windows с требованием выделения оконного ресурса и, если требование удовлетворено, получает так называемый *дескриптор окна*. Класс TWinControl и его потомки хранят дескриптор окна в свойстве Handle. Программист может использовать этот дескриптор для непосредственного обращения к API-функциям Windows. Потомки TGraphicControl не требуют от Windows дефицитного оконного ресурса, но они и не могут получать и обрабатывать Windows-сообщения — управляет такими элементами оконный компонент-владелец (например, форма), который является посредником между Windows и неоконными компонентами.

Имена и владельцы компонентов

Класс TComponent включает в себя свойства и методы, общие для всех компонентов.

Свойство Name определяет имя компонента:

```

property Name: TComponentName;
type TComponentName = String;           // 32-разрядные версии
                                           // Delphi
type TComponentName = String[63]; // Delphi 1
  
```

Имя компонента строится по тем же правилам, что и имена любых других объектов программирования — констант, переменных, подпрограмм и т. д. Оно представляет собой правильный идентификатор и должно быть уникальным в области своей видимости в программе. Так как компоненты помещаются на форму среды Delphi, каждый компонент автоматически получает создаваемое средой имя, совпадающее с именем своего класса (без начальной буквы T) и дополненное числовым суффиксом: Form1, Label1, Edit2 и т. д. Впоследствии программист может переименовать компонент, чтобы сделать текст программы более понятным. При разработке собственных имен полезно выработать свою систему двух- или трехбуквенных префиксов, кодирующих тип компонента. Например, fm — для формы

TForm, lb — для метки TLabel, ed — для поля TEdit и т. п. Комбинируя префикс с мнемоническим именем, можно существенно улучшить читабельность программы и усилить контроль за правильным использованием свойств и методов компонента. Например, показанный ниже оператор сразу вызывает подозрение:

```
lbOutput.Text := 'Текст';
```

Причина в том, что префикс lb определяет компонент TLabel, который не имеет свойства или поля с именем Text.

Следующее свойство определяет произвольный целочисленный параметр, который не используется Delphi и которым программист может распоряжаться по своему усмотрению:

```
property Tag: Integer;
```

Любой компонент Delphi (кроме форм и модулей данных) является собственностью другого компонента и, в свою очередь, может быть владельцем одного или нескольких компонентов. Такая зависимость компонентов друг от друга позволяет существенно упростить процесс управления ими. Например, чтобы удалить окно с включенными в него компонентами, достаточно вызвать деструктор окна: он поочередно вызовет деструкторы всех других компонентов, владельцем которого является компонент-окно, и таким образом полностью освободит выделенные окну ресурсы. Указывает на владельца компонента следующее свойство (это свойство доступно только для чтения):

```
property Owner: TComponent;
```

Положение компонента в массиве Components своего владельца определяет свойство ComponentIndex:

```
property ComponentIndex: Integer;
```

Сам массив содержит список всех компонентов, которыми владеет данный компонент, и определяется следующим свойством:

```
property Components[Index: Integer]: TComponent;
```

Количество зарегистрированных в списке компонентов возвращает свойство ComponentCount:

```
property ComponentCount: Integer;
```

Конструктор TComponent.Create имеет единственный параметр обращения, в котором компоненту передается ссылка на его владельца:

```
constructor Create(AOwner: TComponent);
```

В ходе выполнения конструктора компонент вставляет ссылку на себя в список Components своего владельца и изменяет содержимое собственного свойства Owner.

Для управления списком Components предназначены методы, перечисленные в табл. 14.1.

Таблица 14.1. Методы управления списком Components

Метод	Описание
procedure DestroyComponents;	Вызывает поочередно деструкторы всех компонентов из списка Components и удаляет все компоненты из списка

продолжение ➤

Таблица 14.1 (продолжение)

Метод	Описание
function FindComponent (const AName: String): TComponent;	Отыскивает в списке компонент с именем AName и возвращает ссылку на него
procedure InsertComponent (AComponent: TComponent);	Вставляет компонент AComponent в конец списка Components
procedure RemoveComponent (AComponent: TComponent);	Удаляет компонент AComponent из списка

Родительские и дочерние компоненты

Наследники класса TControl образуют всю палитру видимых компонентов Delphi. В терминах Windows они называются элементами управления, так как на их основе прежде всего реализуются элементы интерфейса Windows — кнопки, флажки, переключатели, списки и т. п. В тексте книги я часто буду употреблять слова компонент и элемент управления как синонимы.

Как уже отмечалось, некоторые из наследников TControl обладают дескрипторами окон и способны получать и обрабатывать Windows-сообщения, другие окон не имеют, но обязательно включаются в состав оконных компонентов, которые управляют ими, согласуясь с требованиями (сообщениями) Windows. Оконные элементы управления обладают специальной оконной процедурой, в которую Windows посылает управляющие сообщения (например, извещения о манипуляции пользователем мышью или о нажатии клавиш). В терминах Windows такие элементы называются родительскими, а связанные с ними неоконные элементы — дочерними. Замечу, что оконный компонент может выступать как родительский не только по отношению к неоконным компонентам, но и к оконным. В этом случае он просто транслирует управляющие Windows-сообщения в оконные процедуры дочерних компонентов. Обязательным требованием Windows является визуальная синхронизация дочерних элементов: они не могут выходить за границы своего родителя и появляются и исчезают вместе с ним. Иными словами, родитель с дочерними элементами рассматривается Windows как единое целое. Класс TControl определяет свойство Parent, которое содержит ссылку на родительский компонент:

property Parent: TWinControl;

Это свойство не следует путать со свойством владения (Owner): владелец *создает* компонент (не обязательно видимый), а родитель (Parent) *управляет видимым* компонентом. Поскольку конструктор TComponent.Create не изменяет свойства Parent (в родительском классе TComponent такого свойства нет), при создании видимых компонентов на этапе прогона программы это свойство необходимо изменять программно. Например, следующий обработчик события OnCreate формы Form1 вставит надпись Дочерний элемент в левый верхний угол формы:

```
procedure TForm1.FormCreate(Sender: TObject);
var
```

```

lbLabel: TLabel;
begin
  lbLabel := TLabel.Create(Self);
  lbLabel.Parent := Self;
  lbLabel.Caption := 'Дочерний элемент';
end;

```

Обратите внимание на следующий оператор:

```
lbLabel.Parent := Self;
```

Если убрать этот оператор из приведенного выше фрагмента, метка никогда не «узнает» о том, что пришла пора прорисовать себя на экране, и ее текст не появится. Наоборот, изменение свойства `Parent` подключает метку к списку дочерних элементов формы, и оконная процедура формы обратится к нужному методу метки, чтобы заставить ее появиться на экране в момент появления самой формы.

ПРИМЕЧАНИЕ Приведенный выше пример является образчиком того, как *не надо* программировать: локальная переменная `lbLabel` будет удалена после выхода из процедуры `FormCreate`, содержащийся в ней указатель будет безвозвратно потерян для программы, и она никогда не сможет освободить связанные с меткой ресурсы памяти. Хотя ресурсы будут автоматически освобождены при завершении работы программы, правильнее было бы расположить переменную `lbLabel` вне процедуры, например, в секции **public** класса `TForm1`, чтобы в обработчике события `OnDestroy` формы выполнить вызов деструктора метки.

Помимо свойства `Components` каждый оконный компонент получает от своего родителя `TWinControl` свойство `Controls`, содержащее список дочерних элементов:

```
property Controls[Index: Integer]: TControl;
```

Свойство `ControlCount` возвращает количество дочерних элементов (длину массива `Controls`):

```
property ControlCount: Integer;
```

Работать со списком `Controls` можно с помощью методов, перечисленных в табл. 14.2.

Таблица 14.2. Методы управления списком `Controls`

Метод	Описание
function <code>ContainsControl(Control: TControl): Boolean;</code>	Возвращает значение <code>True</code> , если компонент <code>Control</code> является дочерним элементом
function <code>ControlAtPos(const Pos: TPoint; AllowDisabled: Boolean): TControl;</code>	Возвращает ссылку на компонент, которому принадлежит точка с координатами <code>Pos</code> . Если <code>AllowDisabled=True</code> , поиск ведется также среди запрещенных для выбора компонентов

— продолжение ↗

Таблица 14.2 (продолжение)

Метод	Описание
function FindNextControl (CurControl: TWinControl; GoForward, CheckTabStop, CheckParent: Boolean): TWinControl;	Ищет новый элемент в списке Controls. CurControl определяет начальную позицию поиска; GoForward=True, если поиск идет от начальной позиции к концу списка; CheckTabStop=True, если поиск идет только среди компонентов со свойством TabStop=True; если CheckParent=True, поиск идет только среди оконных элементов
procedure GetTabOrderList (List: TList);	Создает список List всех дочерних компонентов в порядке их обхода, реализуемого клавишей Tab
procedure InsertControl (AControl: TControl);	Вставляет новый дочерний элемент в список Controls. Программа не должна обращаться непосредственно к этому методу: для вставки дочернего элемента следует установить нужное значение свойства Parent во вставляемом компоненте
procedure NotifyControls (Msg: Word);	Посылает сообщение Msg во все дочерние оконные компоненты
procedure Repaint; и все его дочерние компоненты	Перерисовывает родительский компонент
procedure ScaleControls (M, D: Integer);	Перерисовывает все дочерние элементы с размерами, измененными в M/D раз по сравнению с предыдущими размерами
procedure SelectFirst;	Выбирает первый в порядке обхода, реализуемого клавишей Tab, дочерний компонент
procedure SelectNext (CurControl: TWinControl; GoForward, CheckTabStop: Boolean);	Выбирает очередной в порядке обхода, реализуемого клавишей Tab, дочерний компонент. CurControl определяет начальную точку поиска; GoForward=True, если поиск идет к концу списка; CheckTabStop=True, если выбираются только компоненты со свойством TabStop=True
procedure SetChildOrder (Child: TComponent; Order: Integer);	Устанавливает новый порядок обхода, реализуемого клавишей Tab, компонента Child
procedure ShowControl (AControl: TControl);	Показывает дочерний компонент AControl

Положение, размеры и оформление компонентов

Положение и размеры компонента (в пикселах) определяются четырьмя его свойствами:

```
property Height: Integer; // Высота
property Left: Integer;   // Положение левой кромки
property Top: Integer;    // Положение верхней кромки
property Width: Integer;  // Ширина
```

Для всех компонентов, кроме форм, эти свойства задаются в координатах клиентской части родительского компонента. Для формы — в координатах экрана. Клиентская часть компонента — это внутренняя его область за исключением заголовка, рамки и меню. Свойства обычно определяются на стадии конструирования формы, но они доступны также и на этапе прогона программы. Изменение любого из них приводит к немедленному изменению положения или размера компонента как на этапе конструирования, так и при прогоне программы. Как показано ниже, все четыре числовые величины содержатся также в единственном свойстве:

```
type
TPoint = record
  X: LongInt;
  Y: LongInt;
end;
property BoundsRect: TRect;
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;
```

Это свойство удобно использовать при прорисовке компонента методом `TCanvas.FrameRect`.

В некоторых случаях бывает необходимо пересчитать относительные координаты точки внутри клиентской части в абсолютные координаты экрана и наоборот. Эта задача решается двумя методами `TControl`:

```
function ClientToScreen(const Point: TPoint): TPoint;
function ScreenToClient(const Point: TPoint): TPoint;
```

Важную роль играет свойство `Align`, определяющее положение компонента относительно границ своего родителя:

```
type TAlign = (alNone, alTop, alBottom,
               alLeft, alRight, alClient);
property Align: TAlign;
```

Если это свойство не равно `alNone`, компонент прижимается к верхней (`alTop`), нижней (`alBottom`), левой (`alLeft`) или правой (`alRight`) границе своего родителя. При этом размеры компонента по соседним с границей измерениям

игнорируются, и компонент «растекается» по границе. Например, если `Align=alTop`, значения свойств `Left` и `Width` компонента игнорируются, и его прямоугольник будет занимать всю верхнюю часть клиентской области родителя высотой `Height` пикселей; если `Align=alLeft`, свойства `Top` и `Height` игнорируются и прямоугольник занимает левую часть родителя шириной `Width` пикселей и т. д. Если несколько компонентов имеют одинаковое выравнивание, они последовательно прижимаются друг к другу в порядке их перечисления в свойстве `Controls`: первый прижимается к границе родителя, второй — к границе первого и т. д. Вся не заполненная другими компонентами клиентская область родителя заполняется компонентами со свойствами `Align=alClient`, которые в этом случае (то есть если их несколько) накладываются друг на друга. Замечательной особенностью свойства является его постоянство при изменении размеров клиентской части родителя. Если, например, компонент прижат к верхней границе формы, он будет неизменно занимать верхнюю часть клиентской области при любых изменениях размеров окна. Таким способом можно легко создать панели инструментов, строки состояния и т. п. Временное отключение и затем включение эффекта от свойства `Align` обеспечивается методами

```
procedure DisableAlign;
procedure EnableAlign;
```

Любой видимый компонент можно скрыть или показать с помощью свойства `Visible` или методами `Hide` и `Show`:

```
property Visible: Boolean; // True - показывает
procedure Hide;           // Скрывает компонент
procedure Show;           // Показывает компонент
```

Скрытый компонент не реагирует на события мыши и клавиатуры, он не участвует в «дележе» клиентской области родителя, ему нельзя передать фокус ввода клавишей `Tab`.

Если компонент частично или полностью перекрывается другими компонентами, его можно расположить над всеми компонентами и убрать под них с помощью следующих методов:

```
procedure BringToFront; // Сделать верхним
procedure SendToBack;   // Сделать нижним
```

Свойство `Enabled` определяет возможность активизации компонента:

```
property Enabled: Boolean;
```

Если это свойство имеет значение `False`, компонент недоступен. Такие компоненты, точнее, надписи на них, обычно отображаются серым цветом.

Некоторые компоненты имеют плоское представление (например, метка `TLabel`), другие — всегда объемное (например, кнопка `TButton`). Для остальных элементов объемность представления регулируется свойством `Ctl3D`:

```
property Ctl3D: Boolean;
```

Все потомки класса `TWinControl` получают от своего родителя 4 свойства, которые позволяют очерчивать оконные компоненты внутренней и внешней рамками¹:

¹ Класс `TWinControl` имеет эти свойства только в версиях Delphi 5...7.

```

type
  TBevelEdge = (beLeft, beTop, beRight, beBottom);
  TBevelEdges = set of TBevelEdge;
property BevelEdges: TBevelEdges; // Стороны с рамками

type TBevelKind = (bkNone, bkTile, bkSoft, bkFlat);
property BevelKind: TBevelKind; // Стиль рамок

type TBevelCut = (bvNone, bvLowered, bvRaised, bvSpace);
property BevelInner: TBevelCut; // Внутренняя рамка
property BevelOuter: TBevelCut; // Внешняя рамка

```

Очерчиваются только те стороны компонента, которые содержатся в свойстве BevelEdges. Свойство BevelKind определяет стиль рамок:

- bkNone — рамок нет;
- bkTile — рамки контрастные (черные);
- bkSoft — рамки пониженной контрастности (серые);
- bkFlat — рамки неконтрастные (белые).

Тип TBevelCut определяет эффект объемности для соответствующей рамки:

- bvNone — нет объемности;
- bvLowered — вдавленная рамка;
- bvRaised — выпуклая рамка;
- bvSpace — рамка белая, если свойство BevelKind не содержит bkTile, в противном случае — выпуклая.

Замечу, что наличие описанных свойств у класса TWinControl еще не означает, что они будут у каждого его потомка. Например, свойства BevelEdges и BevelKind при объявлении класса TPanel не переобъявляются, и потому у панели отсутствуют, зато объявляется дополнительное свойство BevelWidth, определяющее толщину рамок.

С каждым управляющим компонентом связывается текстовая строка, которая становится доступна либо через свойство Caption, либо через свойство Text (альтернативой свойству Text, которое имеет тип **String**, является свойство WindowsText типа PChar класса TControl). Независимо от того, какое свойство хранит эту строку, ее можно установить и получить соответственно методами SetTextBuf и GetTextBuf, при этом метод GetTextLen возвращает длину строки:

```

procedure SetTextBuf(Buffer: PChar);
function GetTextBuf(Buffer: PChar; BufSize: Integer):
  Integer;
function GetTextLen: Integer;

```

Если эта строка прорисовывается в компоненте, используется шрифт, задаваемый свойством Font:

```

property Font: TFont;

```

В этом случае свойство TAlignment регулирует расположение текста относительно границ компонента:

```
type TAlignment = (taLeftJustify, taRightJustify,
    taCenter);
```

```
property Alignment: TAlignment;
```

Допустимые значения свойства:

- ✻ taLeftJustify — прижать к левой границе;
- ✻ taRightJustify — прижать к правой границе;
- ✻ taCenter — расположить по центру.

Рассмотрим следующее свойство:

```
property DesktopFont: Boolean;
```

Придав этому свойству значение True, можно потребовать от компонента, чтобы он выводил текст системным шрифтом Windows.

Видимая часть элемента заливается цветом Color:

```
property Color: TColor;
```

Обычно значение этого свойства выбирается в таблице стандартных цветов Windows в виде константы clXXXX (перечень этих констант содержит раскрывающийся список свойства). В некоторых случаях может потребоваться залить компонент нестандартным цветом. В этом случае учтите, что свойство Color хранит четырехбайтовое значение, каждый байт которого (слева направо, то есть от старшего к младшему) имеет следующее назначение:

- ✻ 1 — указатель формата цвета (см. ниже);
- ✻ 2, 3, 4 — интенсивность соответственно синей, зеленой и красной составляющих.

Например, значение \$00000000 определяет черный цвет, \$00FF0000 — чистый синий цвет, \$00FFFFFF — белый цвет и т. д.

Старший байт указывает, как используются остальные байты значения. Если он равен нулю, они определяют RGB-цвет так, как это описано выше. Если старший байт равен 1, три оставшихся байта определяют номер одной из 65 536 возможных логических палитр (второй байт в этом случае игнорируется). Наконец, если старший байт равен 2, младшие определяют относительный цвет: в этом случае Windows отыскивает в текущей логической палитре ближайший к указанному цвет и использует его для заливки компонента. Другие значения старшего байта игнорируются (точнее, игнорируются старшие 5 разрядов этого байта; самый старший бит, если он равен 1, определяет черный цвет независимо от значений остальных 31 разрядов).

Чтобы разнообразить унылые серые окна программы, соответствующие стандартным формам Delphi (в Windows 98), я иногда использую значение \$00AAFFFF, определяющее бледно-желтый цвет (в свойство Color можно вводить значения в виде десятичных или шестнадцатеричных чисел). Получить шестнадцатеричное значение того или иного цвета вам поможет несложная программа (Chap_14\ColorDef\ColorDef.dpr), окно которой показано на рис. 14.2. С ее помощью можно выбрать цвет в окне компонента TColorDialog и получить его шестнадцатеричное представление или ввести шестнадцатеричное число и увидеть соответствующий ему цвет.

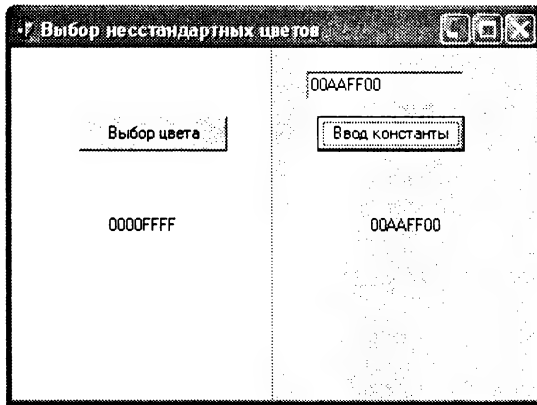


Рис. 14.2. Окно программы для определения цветовых констант

Порядок разработки программы описан ниже.

1. Поместите на пустую форму две панели — они будут наглядно демонстрировать цвет, выбранный с помощью компонента `TColorDialog` (левая панель), и цвет, заданный с помощью шестнадцатеричной константы (правая панель). Установите в свойство `Align` панели `Panel1` значение `alLeft`, а в такое же свойство `Panel2` — значение `alClient`.
2. На левую панель поместите кнопку `TButton` и диалоговое окно `TColorDialog` (этот компонент находится на вкладке `Dialogs` палитры компонентов). В качестве значения свойства `Caption` кнопки введите строку `Выбор цвета`.
3. На правую панель поместите поле `TEdit` и кнопку `TButton`. В свойство `CharCase` поля введите значение `ecUpperCase` — это заставит компонент преобразовывать вводимые в нем буквы в прописные и упростит их дальнейший анализ. В свойство `Caption` кнопки введите строку `Ввод константы`.

Сама программа выбора нестандартных цветов представлена в листинге 14.1.

Листинг 14.1. Выбор нестандартных цветов¹

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
```

¹ Все представленные в книге тексты программ можно загрузить с сайта издательства по адресу <http://www.piter.com/download> — Примеч. ред.


```

    Panel2: TPanel;
    ColorDialog1: TColorDialog;
    Button1: TButton;
    Button2: TButton;
    Edit1: TEdit;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
    { Вызываем метод Execute диалогового окна ColorDialog1
    и контролируем результат: если метод возвращает True,
    пользователь выбрал нужный цвет: }
    if ColorDialog1.Execute then
        begin
            // Закрашиваем левую панель выбранным цветом:
            Panel1.Color := ColorDialog1.Color;
            { Помещаем в поле Edit1 и в заголовок левой панели
            шестнадцатеричное представление цвета: }
            Edit1.Text := IntToHex(ColorDialog1.Color,8);
            Panel1.Caption := Edit1.Text
        end
    end;

    { Следующая функция преобразует шестнадцатеричное
    представление S целого числа в тип Integer }
    function HexStrToInt(S: String): Integer;
    function HDigital(C: Char): Byte;
    // Преобразует шестнадцатеричную цифру C в число
    begin
        case C of
            '0'..'9': Result := ord(C)-ord('0');
            'A'..'F': Result := ord(C)-ord('A')+10;
        else

```

```

{ Если очередной символ не относится к допустимым для
представления шестнадцатеричной цифры, возбуждаем
исключительную ситуацию, которая блокирует преобразование
и сообщает пользователю об ошибке: }
    raise EConvertError.Create
    ('Недопустимое представление шестнадцатеричного числа');
end;
end; // HDigital
var
    k: Integer;
begin // HexStrToInt
    Result := 0;
    // Удаляем возможный символ-указатель шестнадцатеричного
    // числа:
    if S[1]='$' then delete(S,1,1);
    // Цикл преобразования строки:
    for k := 1 to Length(S) do
        Result := Result*16+HDigital(S[k]);
    end; // HexStrToInt

procedure TForm1.Button2Click(Sender: TObject);
begin
    Panel2.Color := HexStrToInt(Edit1.Text);
    Panel2.Caption := Edit1.Text
end;

end.

```

В обработчике события щелчка на кнопке правой панели нужно преобразовать введенное в поле шестнадцатеричное представление цветовой константы в целое число и поместить его в свойство `Caption` правой панели. Поскольку нет стандартной функции преобразования символьного представления шестнадцатеричной константы в число, обработчик использует вспомогательную функцию `HexStrToInt`.

Для упрощения разработки интерфейсных элементов, выдержанных в едином стиле, в состав видимых компонентов включены свойства, связывающие ту или иную особенность оформления или поведения компонента с соответствующей особенностью родителя. Таких свойств четыре:

```

property ParentColor: Boolean;
property ParentCtl3d: Boolean;
property ParentFont: Boolean;
property ParentHint: Boolean;

```

Если, например, цвет компонента не задан явно свойством `Color` и его свойство `ParentColor` имеет значение `True`, компонент будет использовать при рисовке цвет своего родителя, а изменение цвета родителя приведет к немедленному изменению цвета дочернего компонента. Явная установка значений свойств

Color, Ctl3D, Font или Hint приводит к автоматической установке значений False в соответствующие свойства ParentXXX.

В версиях Delphi 4–7 в класс TControl введено свойство, определяющее способ фиксирования дочернего компонента относительно границ контейнера, в котором он размещен:

```
property Anchors: TAnchors;  
type TAnchors = set of TAnchorKind;  
type TAnchorKind = (akTop, akLeft, akRight, akBottom);
```

По умолчанию это свойство имеет значение [akTop, akLeft], что означает фиксацию компонента относительно левого верхнего угла контейнера. В результате возможные изменения размеров контейнера никак не скажутся на изменении положения и/или размеров компонента. Если установить значение [akRight, akBottom], правая и нижняя кромки компонента всегда будут располагаться на одном и том же расстоянии от соответствующих кромок контейнера. Так как положение левого верхнего угла компонента при этом не фиксируется, он будет «плавать» внутри контейнера, всегда располагаясь на одном и том же расстоянии от его правой и нижней кромок. Однако если установлено значение [akTop, akLeft, akRight, akBottom], левый верхний угол фиксируется, и компонент будет пытаться отслеживать расстояние до правого нижнего угла контейнера за счет изменения своих размеров. Поместите на пустую форму поле TEdit и поэкспериментируйте с его свойством Anchors, каждый раз изменяя мышью размеры формы (не обязательно запускать программу — свойство Anchors работает и на этапе конструирования).

В Delphi 4–7 в класс TControl введено свойство Constraints:

```
property Constraints: TSizeConstraints;
```

Это свойство используется для ограничения максимальных и минимальных размеров элементов управления по высоте и ширине. Объект класса TSizeConstraints имеет четыре свойства: MaxHeight, MaxWidth, MinHeight, MinWidth типа 0..MaxInt, которые и определяют предельные значения указанных параметров. По умолчанию эти свойства имеют нулевые значения, что эквивалентно снятию каких бы то ни было ограничений.

Свойство AutoSize также впервые введено в Delphi 4:

```
property AutoSize: Boolean;
```

Это свойство связано в основном с интерфейсом «причаливания» (Drag&Dock) и разрешает (True) или запрещает (False) оконному компоненту автоматически изменять свои размеры в зависимости от количества и размеров содержащихся в нем неоконных компонентов.

Указатели мыши

При перемещении указателя мыши по экрану он может менять свою форму в зависимости от значения свойства Cursor компонента, на котором он расположен в данный момент:

```
property Cursor: TCursor;  
type TCursor = -32768..+32767;
```

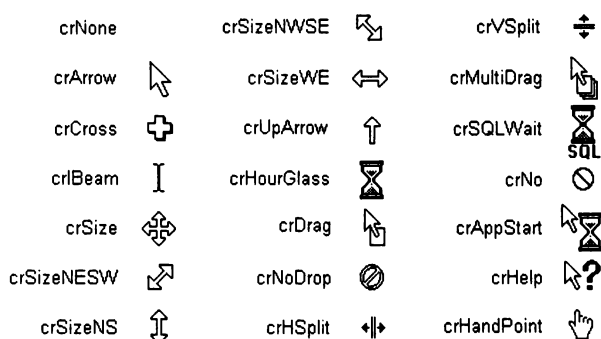


Рис. 14.3. Стандартные указатели Delphi

В Delphi predeterminedены стандартные указатели, показанные на рис. 14.3.

В практике программирования часто возникает необходимость изменения формы указателя для всех окон программы. Например, при выполнении достаточно длительного по времени процесса указатель мыши часто принимает вид песочных часов (`crHourGlass`), а после завершения процесса — восстанавливает свой первоначальный вид. Чтобы изменить форму указателя для всех окон программы одновременно, используется свойство `Cursor` глобального объекта `Screen`, который автоматически создается для каждой программы:

```
Screen.Cursor := crHourGlass;
...
Screen.Cursor := crDefault; // Делаем длительную работу
                           // Восстанавливаем начальную
                           // форму указателя
```

Программист может создать и использовать нестандартный указатель. При этом он должен придерживаться следующей последовательности действий.

1. С помощью редактора изображений Delphi создайте изображение указателя и разместите это изображение в дополнительном ресурсном файле программы.
2. В работающей программе сначала загрузите указатель из ресурсного файла с помощью функции `LoadCursor` и зарегистрируйте его в списке `Cursors` объекта `Screen`.
3. Только после выполнения предыдущих действий можно назначать нестандартный указатель свойствам `Cursor` компонентов или экрана.

Рассмотрим описанный выше процесс на следующем примере, в котором создается и используется указатель в виде окружности.

1. Выберите команду **Tools** ► **Image Editor**, чтобы загрузить редактор изображений Delphi.
2. В окне редактора сначала выберите команду **File** ► **New** ► **Resource File**, а затем — команду **Resource** ► **New** ► **Cursor**. Окно редактора к этому моменту должно иметь вид, показанный на рис. 14.4.
3. С помощью команды **Resource** ► **Rename** обязательно измените стандартное имя ресурса `Cursor1` на любое другое, *набрав его прописными буквами*, например, `MYCURSOR`. Вы можете даже оставить прежнее имя, но набрать его прописными

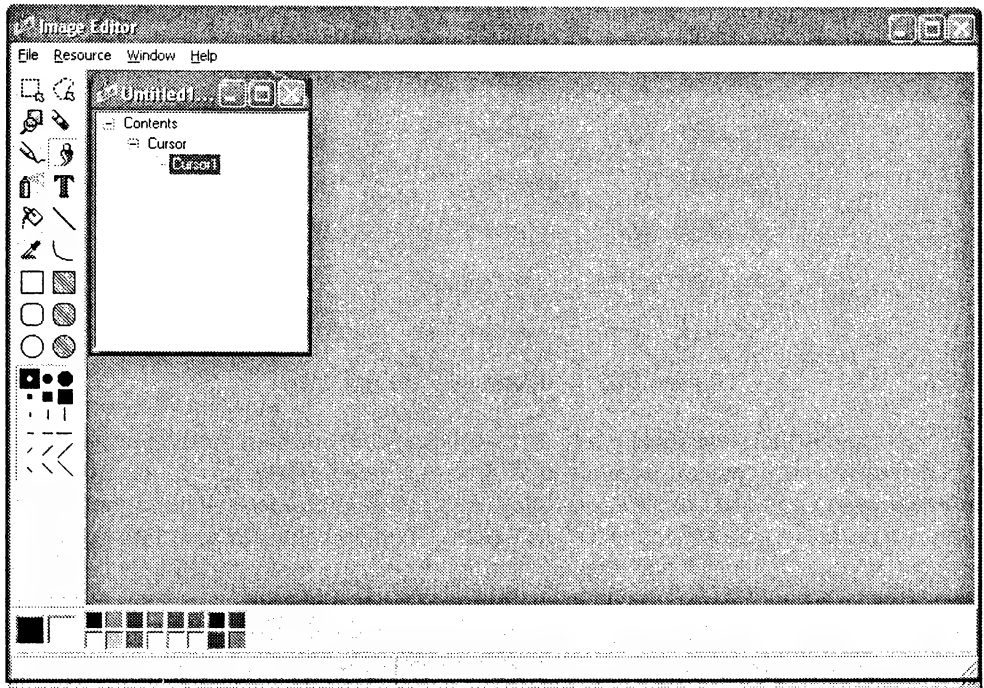


Рис. 14.4. Окно редактора изображений с заготовкой для указателя мыши

буквами нужно обязательно, иначе функция `LoadCursor` не сможет извлечь указатель из ресурсного файла.

4. Дважды щелкните на ресурсе в списке ресурсов или выберите команду **Resource** ► **Edit** — на экране появится окно редактора с двумя пустыми полями, обведенными красными рамками. Левое поле предназначено для представления создаваемого указателя в увеличенном масштабе, правое показывает его в натуральную величину.
5. Выберите инструмент в форме окружности в левой части окна редактора, нажмите и удерживайте клавишу **Shift**, чтобы получилась правильная окружность, подведите указатель мыши в виде крестика несколько правее и ниже верхнего левого угла левого поля редактора, нажмите левую кнопку мыши и, удерживая кнопку мыши и клавишу **Shift**, переместите указатель несколько левее и выше правого нижнего угла поля, после чего отпустите кнопку и клавишу. В результате должно получиться нечто похожее на рис. 14.5.
6. Щелкните мышью на заголовке `Untitled1.res` окна со списком ресурсов (или закройте окно с изображениями указателя кнопкой закрытия в его правом верхнем углу) и с помощью команды **File** ► **Save** сохраните ресурсный файл под именем `Cursor`, после чего закройте редактор изображений.
7. Создайте новый проект с пустой формой и напишите показанный ниже обработчик события `OnCreate` для формы `Form1`:

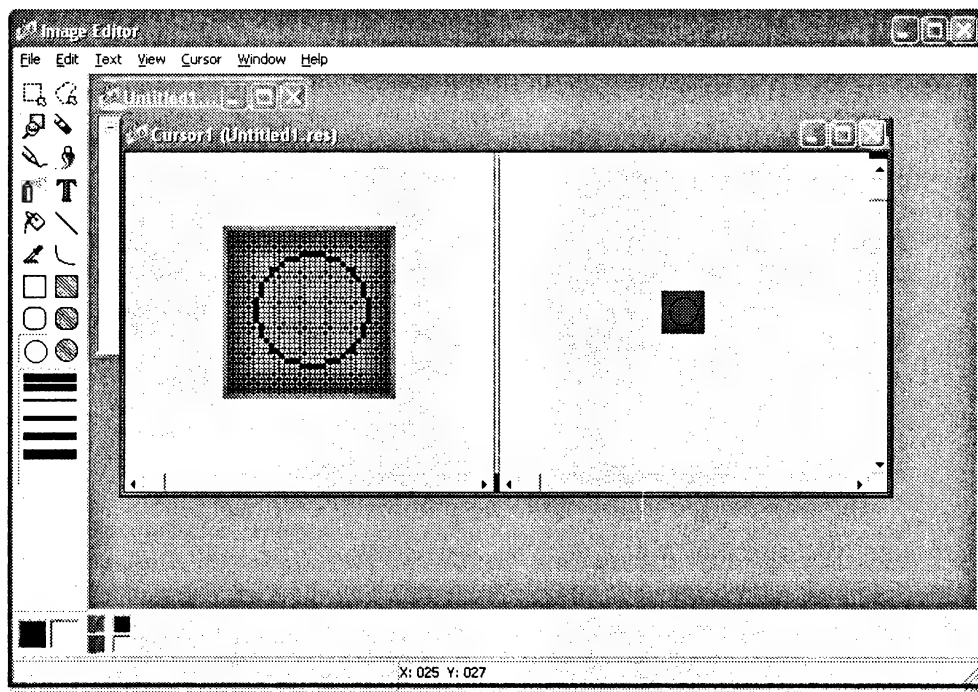


Рис. 14.5. Окно редактора с готовым изображением указателя

```
{ $R *.DFM }
{ $R Cursor.res }

procedure TForm1.FormCreate(Sender: TObject);
begin
    // Регистрируем указатель:
    Screen.Cursors[1] :=
        LoadCursor(HInstance, 'MYCURSOR');
    // Используем его для клиентской части формы:
    Cursor := 1
end;
```

Не забудьте указать комментарий `{ $R Cursor.res }`, чтобы компоновщик присоединил ресурсный файл `Cursor.res` к ресурсному файлу программы. После запуска программы вы увидите в клиентской части пустой формы нестандартный указатель.

Замечу, что при регистрации указателя в списке `Screen.Cursors` индекс может быть любым числом в диапазоне от `-32 768` до `+32 767`. Однако следует учитывать, что индексы в диапазоне от `-22` до `-1` включительно заняты стандартными указателями (см. рис. 14.3), причем индексы `-5` и `-1` соответствуют указателю `crNone`, а индекс `0` всегда связан со стандартным указателем Windows (мышь) и не может переопределяться.

Реакция на события мыши и клавиатуры

События мыши

Для большинства видимых компонентов определен набор обработчиков событий, связанных с мышью:

type

```

TMouseButton = (mbLeft, mbRight, mbMiddle);
TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft,
                      ssRight, ssMiddle, ssDouble);
TMouseEvent = procedure (Sender: TObject;
                          Button: TMouseButton;
                          Shift: TShiftState;
                          X, Y: Integer) of object;
TMouseMoveEvent = procedure (Sender: TObject;
                              Shift: TShiftState;
                              X, Y: Integer) of object;
TNotifyEvent = procedure (Sender: TObject) of object;
property OnMouseDown: TMouseEvent;
property OnMouseUp: TMouseEvent;
property OnMouseMove: TMouseMoveEvent;
property OnClick: TNotifyEvent;
property OnDblClick: TNotifyEvent;
```

Тип `TMouseButton` определяет одну из трех кнопок мыши: левую (`mbLeft`), правую (`mbRight`) и среднюю (`mbMiddle`).

Тип `TShiftState` содержит признаки, уточняющие обстоятельства возникновения события:

- `ssShift` — нажата клавиша `Shift`;
- `ssAlt` — нажата клавиша `Alt`;
- `ssCtrl` — нажата клавиша `Ctrl`;
- `ssLeft` — нажата левая кнопка мыши;
- `ssRight` — нажата правая кнопка мыши;
- `ssMiddle` — нажата средняя кнопка мыши;
- `ssDouble` — нажаты одновременно левая и правая кнопки мыши.

Обработчики событий `OnMouseDown` и `OnMouseUp` определяют реакцию программы на соответственно нажатие и отпускание кнопки мыши, `OnMouseMove` — на перемещение указателя мыши на компоненте, `OnClick` и `OnDblClick` — соответственно на щелчок и двойной щелчок левой кнопкой мыши. Во всех обработчиках параметр `Sender` содержит ссылку на компонент, на котором находился указатель мыши в момент наступления события, а `X` и `Y` определяют координаты точки чувствительности указателя мыши в этот момент в системе координат клиентской области родительского компонента. Замечу, что событие `OnClick` возникает после события `OnMouseDown`, но перед событием `OnMouseUp`, а событие `OnDblClick` возникает после события `OnMouseUp`.

События клавиатуры

События мыши возникают в любом потомке класса `TControl`. В отличие от этого события клавиатуры возникают только в некоторых оконных компонентах (в потомках класса `TWinControl`). Обработка событий связана со следующими свойствами этих компонентов:

type

```
TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft,
                      ssRight, ssMiddle, ssDouble);
```

```
TKeyEvent = procedure (Sender: TObject;
                       var Key: Word;
                       Shift: TShiftState) of object;
```

```
TKeyPressEvent = procedure (Sender: TObject;
                            var Key: Char) of object;
```

```
property OnKeyDown: TKeyEvent;
```

```
property OnKeyUp: TKeyEvent;
```

```
property OnKeyPress: TKeyPressEvent;
```

Параметр `Shift`, как и в обработчиках событий от мыши, содержит уточняющие признаки. Параметр `Key` в обработчиках `TKeyEvent` содержит виртуальный код клавиши, а в обработчике `TKeyPressEvent` — ASCII-символ. Обработчики `OnKeyDown` и `OnKeyUp` перехватывают нажатие большинства клавиш клавиатуры, в то время как обработчик `OnKeyPress` — только нажатие алфавитно-цифровых клавиш. Получаемый им символ `Key` учитывает выбранный язык и нажатую клавишу `Shift`.

Виртуальные коды клавиш определены константами `vk_XXX` в файле `SOURCE\RTL\WIN\WINDOWS.PAS` каталога размещения Delphi. Фактически виртуальный код — это просто уникальный числовой идентификатор клавиши. Для буквенно-цифровых клавиш 0–9 и A–Z виртуальный код совпадает с кодом, возвращаемым функцией `ord('X')`, где `X` — соответствующий прописной символ: `ord('0')`, `ord('W')` и т. д. К сожалению, в зоне дополнительных числовых клавиш уникальность кода не обеспечивается для клавиши `Enter`, которой присвоен код 13, как и аналогичной клавише в основной зоне, а также для правых и левых клавиш `Shift`, `Alt` и `Ctrl`. Кроме того, клавиши 0–9 и `Del` в зоне дополнительной клавиатуры сопровождаются уникальным кодом только при активном переключателе `NumLock`, в противном случае они повторяют коды соответствующих основных клавиш. Все остальные клавиши стандартной клавиатуры (за исключением клавиши `Print Screen`, клавиш перемещения курсора и клавиши `Tab`, нажатие на которые не передается в обработчики `TKeyEvent`) имеют постоянно закрепленные за ними числовые коды, позволяющие легко установить факт нажатия или отпускания любой из них.

Поскольку параметр `Key` в каждом обработчике определен как параметр-переменная, программист может изменить фактический код клавиши. Такая возможность может оказаться полезной для фильтрации нажатия каких-либо клавиш. При этом изменение кода происходит в обработчике формы, а в оконный элемент с фокусом ввода будет поступать уже измененный код. Чтобы форма получила сообщение от клавиатуры до передачи его в элемент с фокусом ввода, следует поместить в следующее свойство формы значение `True`:

```
property KeyPreview: Boolean;
```


Клавиатура в Windows

Следует заметить, что Windows значительно «строже» относится к использованию клавиатуры, чем MS-DOS. Это может вызывать проблемы при переносе в среду Delphi игровых приложений, а также приложений, созданных с помощью системы FoxPro или Clipper.

Если вы захотите сохранить устоявшиеся приемы использования клавиатуры в новой разработке, вам, возможно, придется перехватывать сообщения Windows, так как только таким способом программа сможет опознать факт нажатия системных клавиш Alt, Tab, Shift и т. п. Нажатие остальных клавиш можно анализировать путем перехвата сообщений от клавиатуры в обработчиках OnKeyXXX формы при значении True ее свойства KeyPreview.

Например, пусть сочетание клавиш Alt+X используется в существующей программе для закрытия модального диалогового окна. Чтобы разработанное вами окно закрывалось по этой команде, напишите для него следующий обработчик события OnKeyDown:

```
procedure TForm2.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if (Key = ord('X')) and (ssAlt in Shift) then Close
end;
```

Во многих случаях можно использовать собственные клавиши быстрого вызова Windows. К сожалению, такими клавишами снабжаются лишь команды меню, но связанные с ними (командами) обработчики OnClick выполняются даже в том случае, когда команда меню не видна (ее свойство Visible имеет значение False). Этим можно воспользоваться, чтобы вставить в главное меню окна невидимые команды, связав их с нужными клавишами. Пусть, например, кнопка Button1 должна «нажиматься» при нажатии клавиш Ctrl+S. Поместите на форму главное меню (если его еще нет) и создайте в нем пункт с произвольным именем, например, MyButton. В списке свойства ShortCut пункта меню выберите вариант Ctrl+S и установите значение False в его свойство Visible. Теперь можно связать обработчик события OnClick пункта меню непосредственно с событием Button1Click или написать такой обработчик:

```
procedure TForm2.MyButtonClick(Sender: TObject);
begin
  Button1Click(Self)
end;
```

И хотя пункт MyButton главного меню не виден, нажатие связанных с ним клавиш Ctrl+S вызовет срабатывание нужного обработчика. В этом случае форма может не перехватывать клавиатурный ввод.

Как уже отмечалось, обработчики OnKeyXXX не реагируют на нажатие системных клавиш Windows, в том числе клавиш перемещения курсора. Происходит это из-за того, что оконная процедура программы по умолчанию осуществляет стандартную обработку соответствующих сообщений Windows.

ПРИМЕЧАНИЕ В Delphi 5...7 введен специальный компонент TApplicationEvents, который существенно упрощает обработку любых сообщений Windows до того, как они поступят в оконную функцию активной формы.

Для клавиш определены виртуальные коды, перечисленные в табл. 14.3.

ПРИМЕЧАНИЕ В квадратных скобках указаны клавиши из зоны дополнительных цифровых клавиш.

Таблица 14.3. Виртуальные коды клавиш

Код	Значение	Клавиша
vk_Back	8	Backspace
vk_Tab	9	Tab
vk_Clear	12	[5]
vk_Return	13	Enter
vk_Shift	16	Shift
vk_Control	17	Ctrl
vk_Menu	18	Alt
vk_Pause	19	Pause
vk_Capital	20	Caps Lock
vk_Escape	27	Esc
vk_Space	32	Пробел
vk_Prior	33	Page Up
vk_Next	34	Page Down
vk_End	35	End
vk_Home	36	Home
vk_Left	37	←
vk_Up	38	↑
vk_Right	39	→
vk_Down	40	↓
vk_Insert	45	Insert
vk_Delete	46	Delete
vk_0-vk_9	48–57	0–9
vk_A-vk_Z	65–90	A–Z
vk_LWin	91	Windows (левая)
vk_RWin	92	Windows (правая)
vk_Numpad0-vk_Numpad9	96–105	[0]–[9]
vk_Multiply	106	[*]

— продолжение ↗

Таблица 14.3 (продолжение)

Код	Значение	Клавиша
vk_Add	107	[+]
vk_Subtract	109	[-]
vk_Decimal	110	[Del]
vk_Divide	111	[/]
vkF1-vkF12	112-123	F1-F12
vk_Numlock	144	Num Lock
vk_Scroll	145	Scroll Lock
	186	:
	187	+
	188	<
	189	-
	190	>
	191	?
	192	~
	219	[
	220	\
	221	]

Учтите, что константы vk_A-vk_Z и vk_0-vk_9 не определены в файле SOURCE\RTL\WIN\WINDOWS.PAS и, следовательно, в модуле Windows, поэтому для них компилятор выдаст сообщение о неизвестном идентификаторе — в этом случае используйте собственные определения констант или их числовые эквиваленты. Замечу также, что путем анализа параметра Msg.lParam в обработчике Application.OnMessage можно отличить нажатие левых клавиш Alt, Ctrl, Shift от нажатия одноименных правых клавиш, а также нажатие клавиши Enter на основной клавиатуре от нажатия клавиши Enter на дополнительной клавиатуре.

Фокус ввода

Поскольку клавиатура одна, а элементов, могущих ее использовать, может быть много, необходимо каким-то способом выделять элемент, которому в данный момент передается клавиатурный ввод. Это выделение достигается передачей элементу *фокуса ввода* с помощью его метода:

```
procedure SetFocus;
```

Компонент с фокусом ввода имеет значение True в своем свойстве

```
property Focused: Boolean;
```

Если элемент недоступен или невидим, его показанное ниже свойство имеет значение False:

```
property CanFocus: Boolean;
```

Фокус ввода передается элементу после щелчка на нем мышью или в порядке выбора его клавишей Tab. Чтобы элемент можно было выбирать с помощью этой клавиши, следует установить в следующее его свойство значение True:

property TabStop: Boolean;

Порядок выбора элемента определяется другим свойством:

property TabOrder: Integer;

Delphi следит за уникальностью и непрерывностью значений этого свойства у всех помещенных на форму доступных и видимых в данный момент компонентов. С помощью следующего метода можно получить список всех дочерних элементов, выбираемых клавишей Tab:

procedure GetTabOrderList(List: TList);

Механизм действий

В Delphi имеется специальный механизм *действий*, дающий удобное средство централизованной реакции программы на те или иные действия пользователя. Для реализации действий на вкладке Standard палитры компонентов помещен специальный компонент TActionList (см. главу 15), а в класс TComponent введено свойство Action:

property Action: TBasicAction;

Централизация необходима потому, что подавляющее большинство промышленных программ для Windows содержат два или более интерфейсных элемента, реализующих одно и то же действие. Например, в среде Delphi некоторые команды главного меню дублируются соответствующими кнопками панели инструментов (в принципе, *любая* команда меню Delphi может иметь кнопочный эквивалент).

Компонент TActionList содержит список имен действий и связанных с ними имен подпрограмм, которые эти действия реализуют. Чтобы связать компонент с нужным действием, достаточно раскрыть список Action (этот список содержит имена всех действий во всех компонентах TActionList) и выбрать нужное название. При этом обычно меняется графическое изображение и/или надпись на компоненте, а также связанная с ним оперативная подсказка (Hint). Свойство Action используется только после создания и заполнения списка (списков) TActionList.

Учтите, что если для действия не определен обработчик OnExecute, все связанные с этим действием элементы управления на этапе прогона программы будут недоступными.

Механизм перетаскивания

Операционная система Windows широко использует специальный прием связывания программ с данными, который называется Drag&Drop (перетаски и отпусти). Такой прием в Проводнике Windows применяется для копирования или перемещения файлов, а также для запуска обрабатывающей программы. Если, например, файл с расширением DOC перетящить на значок программы Word, автоматически запустится текстовый редактор Word и в его окне появится текст из этого файла.

В Delphi реализован собственный механизм Drag&Drop, позволяющий компонентам обмениваться данными путем перетаскивания их мышью. Этот механизм определяется двумя свойствами и тремя событиями, доступными каждому видимому компоненту.

Первое свойство:

```
TDragMode = (dmManual, dmAutomatic);
```

```
property DragMode: TDragMode;
```

Это свойство определяет, как будет выполняться весь комплекс действий, связанных с перетаскиванием:

- dmManual — вручную (то есть все необходимые для обслуживания механизма события генерируются программой);
- dmAutomatic — автоматически (события инициируются свойствами и методами компонентов).

Во всех случаях программист должен написать обработчики этих событий (см. ниже).

Второе свойство:

```
property DragCursor: TCursor;
```

Это свойство определяет вид указателя мыши в момент, когда на компоненте оказываются перетаскиваемые данные. Если компонент готов принять данные, он устанавливает в это свойство значение `crDrag`, в противном случае — `crNoDrag`. Установка этих свойств осуществляется автоматически, если выполняется условие:

```
DragMode = dmAutomatic
```

Следующее событие возникает, когда указатель мыши «с грузом» оказывается на компоненте.

```
TDragState = (dsDragEnter, dsDragLeave, dsDragMove);
```

```
TDragOverEvent = procedure (Sender, Source: TObject;
```

```
                                  X, Y: Integer; State:
```

```
                                  TDragState;
```

```
                                  var Accept: Boolean) of object;
```

```
property OnDragOver: TDragOverEvent;
```

Параметры здесь следующие:

- Sender — компонент, который генерирует событие (обычно имеет значение Self — сам компонент-получатель, хотя при программном управлении механизмом перетаскивания это может быть не так);
- Source — компонент-отправитель «груза»;
- X, Y — текущие координаты указателя мыши в пикселах клиентской области компонента;
- State — состояние указателя:
 - ☐ dsDragEnter — только что появился на компоненте;
 - ☐ dsDragLeave — только что покинул компонент или была отпущена кнопка мыши;

- `dsDragMove` — перемещается на компоненте;
- ✱ `Accept` — в этом параметре обработчик сообщает, готов ли компонент принять данные (`True` — готов).

Следующее событие означает, что пользователь «бросил» данные на компонент:

```
TDragDropEvent = procedure (Sender, Source: TObject;
                               X, Y: Integer) of object;
property OnDragDrop: TDragDropEvent;
```

Параметры обработчика аналогичны одноименным параметрам обработчика `OnDragOver`.

Наконец, при завершении перетаскивания (вне зависимости от того, приняты данные или нет) возникает еще одно событие:

```
TEndDragEvent = procedure (Sender, Target: TObject;
                              X, Y: Integer) of object;
property OnEndDrag: TEndDragEvent;
```

Здесь `Sender` — отправитель данных; `Target` — получатель данных или `NIL`, если никто не принял «посылку»; `X`, `Y` — координаты мыши в момент отпускания левой кнопки.

Чтобы проиллюстрировать использование механизма перетаскивания, загрузите учебную программу, установите во всех ее компонентах в свойство `DragMode` значение `dmAutomatic` и создайте для метки обработчик события `OnDragOver`, представленный в листинге 14.2 (проект `Chap_14\Drag&Drop\DragDrop.dpr`).

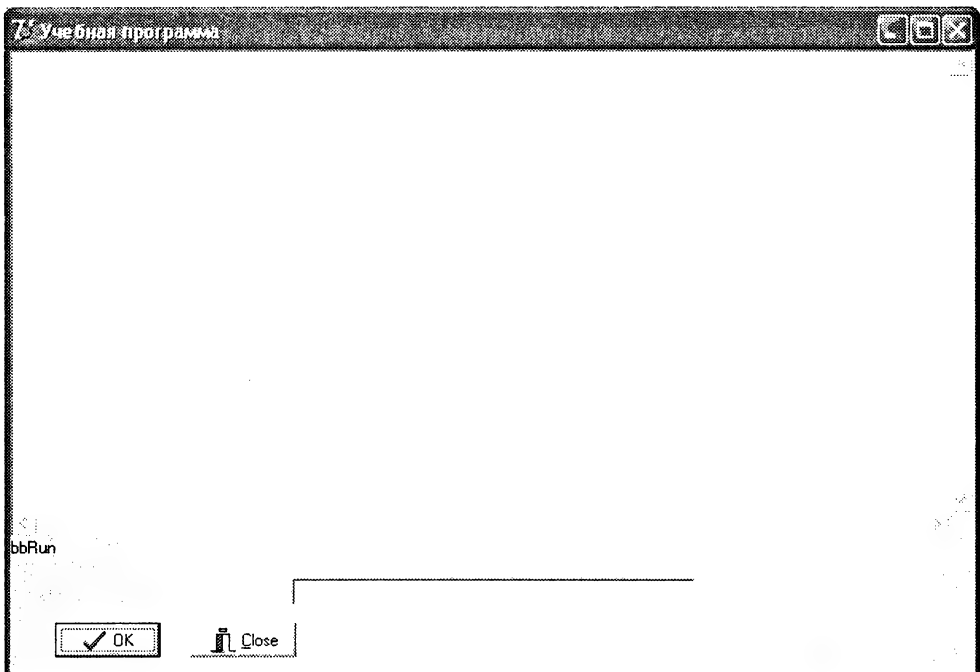


Рис. 14.6. Метка `lbOutput` показывает имя перетаскиваемого компонента

Листинг 14.2. Обработчик события OnDragOver

```

procedure TfmExample.lbOutputDragOver(Sender, Source:
  TObject; X, Y: Integer; State: TDragState; var Accept:
  Boolean);
begin
  Accept := True;
  lbOutput.Caption := (Source as TComponent).Name
end;

```

Теперь перетаскивание любого компонента на метку lbOutput заставит ее показать имя этого компонента (рис. 14.6).

Для ручного (программного) управления механизмом перетаскивания используются специальные методы, доступные любому потомку класса TControl и перечисленные в табл. 14.4.

Таблица 14.4. Методы управления механизмом Drag&Drop

Метод	Описание
procedure BeginDrag (Immediate: Boolean);	Используется источником для инициирования процесса перетаскивания. Если Immediate=True, процесс начинается немедленно, в противном случае — после смещения указателя мыши на 5 пикселей в любом направлении
procedure DragDrop(Source: TObject; X, Y: Integer);	Вызывает обработчик события OnDragDrop
procedure EndDrag(Drop: Boolean);	Вызывает обработчик события OnEndDrag и в параметре Drop сообщает о том, были ли приняты данные

Механизм причаливания

В Delphi введена поддержка специального механизма Drag&Dock (перетащи и причаль), посредством которого можно «причалить» мышью компонент в новое место.

В механизме Drag&Dock участвуют два компонента: принимающий компонент (всегда потомок класса TWinControl) и причаливаемый компонент (потомок класса TControl). Принимающий компонент должен иметь значение True в своем свойстве DockSite. Поскольку причаливание является разновидностью более общего механизма перетаскивания, в класс TControl введено дополнительное свойство, благодаря которому различаются способы использования мыши:

```

property DragKind: TDragKind;
type TDragKind = (dkDrag, dkDock);

```

Значение этого свойства для перетаскивания равно dkDrag, а для причаливания — dkDock (и у причаливаемого, и у принимающего компонентов).

Количество установленных на форме причаливаемых компонентов (то есть компонентов, для которых выполняются условия DragKind=dkDock и DockSite=

=False) определяется свойством `DockClientCount`, а их список хранится в индексированном свойстве `DockClients`.

Для реализации причаливания в класс `TControl` введены некоторые свойства и события, которые перечислены ниже.

property `AutoSize: Boolean;`

Это свойство разрешает (True) или запрещает (False) оконному компоненту менять свои размеры автоматически в зависимости от количества и размеров содержащихся в нем дочерних компонентов.

property `FloatingDockSiteClass: TWinControlClass;`

Это свойство определяет класс окна, в которое будет погружен дочерний элемент вне границ оконного элемента. Неоконные компоненты могут не только причаливаться к оконным компонентам, но и покидать их. После «отчаливания» от компонента неоконные элементы автоматически оказываются в окне, тип которого содержит свойство `FloatingDockSiteClass`. По умолчанию это окно имеет уменьшенный по высоте заголовок и системную кнопку закрытия.

property `DockOrientation: TDockOrientation;`

type `TDockOrientation = (doNoOrient, doHorizontal, doVertical);`

В этом свойстве можно установить (или получить) ориентацию, которую будет иметь «причаливаемый» компонент в окне родителя:

- `doNoOrient` — сохраняется исходная ориентация перемещаемого компонента;
- `doHorizontal` — компонент будет иметь горизонтальную ориентацию;
- `doVertical` — компонент будет иметь вертикальную ориентацию.

property `LRDockWidth: Integer;`

С помощью этого свойства можно получить ширину последнего перемещенного компонента, расположившегося горизонтально.

property `TBDockHeight: Integer;`

С помощью этого свойства можно получить высоту последнего вертикально расположенного компонента.

property `UndockHeight: Integer;`

property `UndockWidth: Integer;`

Эти свойства определяют соответственно высоту и ширину последнего «отчалившего» компонента.

property `Constraints: TSizeConstraints;`

Это свойство с помощью объекта класса `TSizeConstraints` накладывает ограничения на возможные размеры причаливаемого компонента (задает максимальные и минимальные величины для высоты и ширины).

Следующие события аналогичны событиям `OnDragDrop` и `OnDragOver` (см. выше):

property `OnDockDrop: TDockDropEvent;`


```
property OnDockOver: TDockOverEvent;
type TDockOverEvent = procedure (Sender: TObject;
    Source: TDragDockObject; X, Y: Integer; State:
    TDragState; var Accept: Boolean) of object;
```

Следующее событие возникает перед событием OnDragDrop:

```
property OnGetSiteInfo: TGetSiteInfoEvent;
```

Обработчик этого события должен сообщить объекту TDragDockObject, который автоматически связывается с причаливаемым объектом, некоторую дополнительную информацию (размеры, которые будет иметь этот объект, окажется ли он в «плавающем» окне и т. п.).

Осталось еще одно событие:

```
property OnUnDock: TUnDockEvent;
type TUnDockEvent = procedure (Sender: TObject; Client:
    TControl; var Allow: Boolean)
of object;
```

Это событие возникает при «отчаливании» неоконного компонента. Обработчик этого события должен поместить в свойство Allow значение True, если компонент Client может покинуть границы своего владельца Sender.

Все указанные события обрабатываются автоматически, если оба компонента (клиент и сервер) содержат значение dmAutomatic в своем свойстве DragMode.

Чтобы познакомиться с технологией причаливания, поместите на пустую форму панель TPanel и кнопку TButton, установите для панели значение True в свойство DockSite, установите для обоих компонентов значение dkDock в свойстве DragKind и dmAutomatic в свойстве DragMode. После запуска программы перетащите кнопку на панель, а затем стащите ее обратно.

Поддержка справочной службы

Все видимые элементы имеют следующие свойства:

```
property Hint: String;
property ShowHint: Boolean;
```

Эти свойства регулируют появление и содержание *оперативной*, или *всплывающей, подсказки* — небольшого справочного окна возле элемента, на котором останавливается указатель мыши. Оперативные подсказки существенно облегчают начинающему пользователю знакомство с программным продуктом. Введение механизма оперативных подсказок во все видимые компоненты легко решает проблему создания дружелюбного программного интерфейса в современном стиле.

Чтобы компонент смог показать оперативную подсказку, нужно поместить текстовую строку в его свойство Hint и присвоить свойству ShowHint значение True. Обычно оперативная подсказка содержит максимально лаконичный текст, чтобы не занимать слишком большую площадь экрана. Вы можете дополнить этот текст развернутым сообщением в любом компоненте, который способен отображать текст. Как правило, это строка состояния, занимающая нижнюю часть формы. Чтобы отобразить «длинное» сообщение, его нужно, прежде всего, поместить в свойство

Hint сразу после «короткого» и отделить обе части символом вертикальной черты (|). Например:

```
MyControl.Hint := 'Подсказка|Развернутое сообщение';
```

Далее, необходимо в самом начале программы (обычно в обработчике события OnCreate главной формы) указать программе метод, который будет обрабатывать событие OnHint. Это событие возникает в момент появления оперативной подсказки и предназначено для отображения длинного сообщения, которое метод-обработчик может получить в свойстве Hint глобального объекта Application. Например:

```
type
  TForm1 = class (TForm)
    pnStatus: TPanel; // Панель для отображения длинной части
                  // свойства Hint
    ...
  public
    // Объявляем метод-обработчик события OnHint:
    procedure ShowLongHint(Sender: TObject);
    ...
  end;

procedure TForm1.ShowLongHint(Sender: TObject);
{ Этот обработчик помещает на панель pnStatus длинную часть
свойства Hint }
begin
  pnStatus.Caption := Application.Hint;
end;

procedure TForm1.FormCreate(Sender: TObject);
// Метод определяет свойство OnHint объекта-программы
begin
  Application.OnHint := ShowLongHint;
end;
```

Обратите внимание: обработчик события OnHint должен быть методом класса, а не обычной процедурой. Вот почему в предыдущем примере в секцию **public** класса TForm1 вставлено описание метода ShowLongHint. И еще одно замечание. Обычно свойство Hint компонента устанавливается на этапе конструирования формы. В этом случае вы можете создать только однострочную оперативную подсказку. Если же установить свойство Hint на этапе прогона, вы сможете вставить в символьную строку разделители EOLN и таким способом создать многострочную оперативную подсказку:

```
bbClose.Hint := 'Эта кнопка'#13'завершает'
работу'#13'программы';
```

Свойства глобального объекта-программы Application, перечисленные в табл. 14.5, регулируют цвет и время появления оперативной подсказки.

Таблица 14.5. Свойства объекта Application, регулирующие цвет и время появления оперативной подсказки

Свойство	Описание
property HintColor: TColor;	Определяет цвет фона окна оперативной подсказки
property HintHidePause: Integer;	Определяет длительность показа оперативной подсказки в миллисекундах. По умолчанию равно 2500 (2,5 секунды). Чтобы оперативная подсказка вообще не исчезала, установите значение -1
property HintPause: Integer;	Определяет паузу в миллисекундах после остановки указателя мыши и перед появлением оперативной подсказки. По умолчанию равно 500
property HintShortPause: Integer;	Определяет паузу перед появлением оперативной подсказки при переходе с одного компонента с заданным свойством Hint на другой такой же. По умолчанию равно 50

Все видимые компоненты содержат следующее свойство, с помощью которого компонент привязывается к контекстной справочной службе:

property HelpContext: Integer;

В это свойство следует поместить идентификатор раздела файла помощи, который будет автоматически появляться в окне справки после нажатия клавиши F1.

У многих видимых компонентов имеется свойство PopupMenu:

property PopupMenu: TPopupMenu;

Это свойство определяет контекстное меню, которое вызывается щелчком на компоненте правой кнопки мыши.

Поддержка технологии COM

Модель компонентных объектов (Component Object Model, COM) представляет собой технологию обмена объектами между разными приложениями и даже между разными сетевыми машинами. В Delphi включены средства поддержки COM как в виде готовых компонентов (например, DCOMConnection), так и на уровне класса TComponent, то есть относящиеся ко всем компонентам Delphi.

Следующее свойство этого класса возвращает интерфейс COM-объекта, если данный компонент поддерживает технологию COM:

property ComObject: IUnknown;

Если технология COM компонентом не поддерживается, обращение к свойству возбудит исключительную ситуацию EComponentError.

property VCLComObject: Pointer;

Это свойство предназначено для внешнего использования (то есть для экспорта COM-объекта) и предоставляет ссылку на COM-объект Delphi.

Свойства разного назначения

Узнать текущее состояние любого элемента управления на этапе прогона программы можно с помощью следующего свойства класса TControl:

```
type TControlState = set of (csLButtonDown, csClicked,
    csPalette, csReadingState, csAlignmentNeeded, csFocusing,
    csCreating, csCustomPaint, csDestroyingHandle);
property ControlState: TControlState;
```

Элементы множества TControlState имеют следующий смысл:

- csLButtonDown — на элементе была нажата и еще не отпущена левая кнопка мыши;
- csClicked — то же, что csLButtonDown, но устанавливается, если стиль компонента (см. ниже) содержит флаг csClickEvents; означает, что нажатие кнопки мыши интерпретируется как щелчок;
- csPalette — элемент или его родитель получил сообщение WM_PALETTE-CHANGED;
- csReadingState — элемент читает данные из потока;
- csAlignmentNeeded — элемент нуждается в перерисовке, поскольку изменилось его свойство Alignment;
- csFocusing — программа пытается передать элементу фокус ввода;
- csCreating — элемент и/или его родительские и дочерние элементы находятся в стадии создания (флаг очищается после завершения процесса);
- csCustomPaint — элемент перерисовывается;
- csDestroyingHandle — разрушается Windows-дескриптор элемента.

Представленное ниже свойство содержит стилевые флаги элемента:

```
type TControlStyle = set of (csAcceptsControls,
    csCaptureMouse, csDesignInteractive, csFramed,
    csClickEvents, csSetCaption, csOpaque, csDoubleClicks,
    csFixedWidth, csFixedHeight, csNoDesignVisible,
    csReplicable, csNoStdEvents, csDisplayDragImage,
    csReflector, csActionClient, csMenuEvents);
property ControlStyle: TControlStyle;
```

Стилевые флаги имеют следующий смысл:

- csAcceptsControls — элемент может стать получателем дочернего элемента в технологии причаливания;
- csActionClient — элемент может участвовать в централизованном действии;
- csCaptureMouse — элемент будет получать все сообщения от мыши, даже если ее указатель выйдет из его границ;
- csClickEvents — элемент может воспринимать щелчок мыши;
- csDesignInteractive — элемент воспринимает щелчок правой кнопкой мыши на этапе конструирования формы;

- ❖ `csDisplayDragImage` — элемент способен менять свое изображение, когда на нем оказывается перетаскиваемый «груз»;
- ❖ `csDoubleClicks` — элемент может воспринимать двойной щелчок мыши;
- ❖ `csFixedHeight` — элемент имеет фиксированную высоту;
- ❖ `csFixedWidth` — элемент имеет фиксированную ширину;
- ❖ `csFramed` — элемент имеет объемную рамку;
- ❖ `csNoDesignVisible` — элемент не виден на этапе конструирования формы;
- ❖ `csNoStdEvents` — элемент не реагирует на стандартные события мыши и клавиатуры;
- ❖ `csOpaque` — элемент закрашивает всю свою клиентскую область;
- ❖ `csReflector` — элемент способен получать диалоговые сообщения Windows, а также сообщения, связанные с получением/потерей фокуса ввода и изменением размеров (используется для компонентов ActiveX);
- ❖ `csReplicatable` — элемент способен отобразить свой образ методом `PaintTo`;
- ❖ `csSetCaption` — элемент будет изменять свой заголовок при изменении своего свойства `Name`.

Следующие два свойства класса `TWinControl` определяют режим работы и имя специального редактора IME (Input Method Editor) для предварительной обработки клавиатурного ввода в случае использования азиатских языков:

```
type TImeMode = (imDisable, imClose, imOpen, imDontCare,
                  imSAlpha, imAlpha, imHira, imSKata, imKata,
                  imChinese, imSHanguel, imHanguel);
```

```
property ImeMode: TImeMode;
```

```
type TImeName = String;
```

```
property ImeName: TImeName;
```

С помощью следующего свойства класса `TControl` программа может задать порядок чтения текста, расположение вертикальных полос прокрутки и изменение свойства `Alignment` так, чтобы учесть особенности письма справа налево (например арабского):

```
type TBiDiMode = (bdLeftToRight, bdRightToLeft,
                  bdRightToLeftNoAlign, bdRightToLeftReadingOnly);
```

```
property BiDiMode: TBiDiMode;
```

Глава 15

Компоненты вкладки Standard

Описываемые в этой главе компоненты имеют самое общее назначение. Они широко используются при создании любых окон. Многие из них представляют собой компонентную оболочку для популярных объектов интерфейса Windows.

TFrame — фрейм и шаблоны компонентов

Этот компонент определяет фрейм — контейнер для размещения других компонентов. В функциональном отношении компонент почти повторяет свойства формы и отличается от нее в основном лишь тем, что его можно помещать в формы или в другие фреймы. Фактически, фрейм представляет собой удобное средство создания *шаблонов* — произвольных наборов компонентов, максимально приспособленных для нужд конкретного пользователя. Раз созданный шаблон может, подобно любому другому компоненту, размещаться в форме или в другом фрейме (допускается неограниченная вложенность фреймов). Замечу, что любые изменения в базовом фрейме (то есть во фрейме, сохраненном в палитре компонентов) тут же отображаются во всех использующих его проектах.

Первоначально проекту ничего не известно о ранее созданных фреймах, поэтому попытка поместить на пустую форму компонент-фрейм вызовет следующее сообщение (В проекте нет фреймов. Чтобы создать фрейм, выберите команду **File ▶ New ▶ Frame**):

```
No frames in project.
```

```
To create a frame select File | New | Frame.
```

Это сообщение и описываемая ниже методика подключения шаблонов — единственное, что отличает механизм использования шаблонов от использования стандартных компонентов.

Создадим простой фрейм, содержащий две кнопки — `mbOk` и `mbCancel`. Такой шаблон может пригодиться при разработке различных диалоговых окон.

1. Создайте новый проект (**File ▶ New ▶ Application**). Если он все еще связан с учебной формой, определенной в главе 4, предварительно вызовите окно редактора

хранилища объектов (Tools ► Repository), в списке Pages щелкните на пункте Forms, а в списке Objects — на пункте fmExamlpe, после чего снимите флажок Main Form и закройте окно щелчком на кнопке OK.

2. Создайте новый фрейм, выбрав команду File ► New ► Frame.
3. Поместите на фрейм две кнопки TBitBtn (рис. 15.1, слева) и установите следующие свойства для кнопок и фрейма:

```
BitBtn1: Left=0, Top=0, Kind=bkOK, Name=bbOK;
BitBtn2: Left=90, Top=0, Kind=bkCancel, Name=bbCancel;
Frame1 : Height=25, Width=165,
        Anchors=[akRight,akBottom].
```

ПРИМЕЧАНИЕ

Указанные значения свойств создадут минимальный по размерам фрейм с двумя именованными кнопками. Так как свойство Anchors фрейма содержит значения [akRight, akBottoms], фрейм будет постоянно отслеживать свое положение относительно правого нижнего угла контейнера, в котором он окажется.

4. Щелкните на фрейме правой кнопкой мыши и выберите в контекстном меню команду Add To Palette — на экране появится диалоговое окно с предложением сохранить модуль шаблона в файле на диске.
5. Сохраните шаблон в файле с именем DlgBtnsF в папке для ваших программ (см. файл Source\Chap_15\TFrame\DlgBtnF.pas). На экране появится окно регистрации шаблона в палитре компонентов, показанное на рис. 15.1, справа. В окне предлагается задать для нового компонента имя класса и выбрать вкладку палитры компонентов. Обратите внимание на автоматически (по умолчанию) созданное имя заготовки — TFrame1Template. Так как это имя будет появляться в оперативной подсказке Hint при наведении указателя мыши на компонент, следует дать ему более осмысленное имя, например, TDialogButtons. И еще одно замечание — в качестве вкладки палитры компонентов предлагается вкладка Templates (Шаблоны), которая отсутствует в начальном наборе вкладок палитры. Если вы сохраните это имя, в палитре компонентов Delphi будет создана новая вкладка.
6. Задайте имя класса шаблона и выберите вкладку его размещения, после чего щелкните на кнопке OK — шаблон готов для использования.

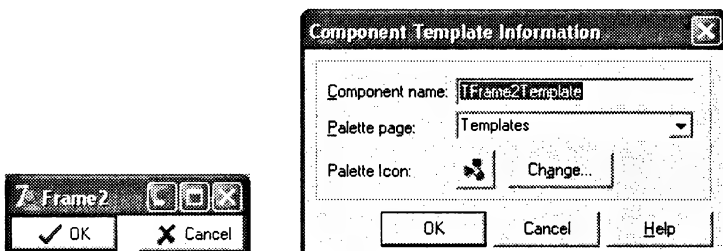


Рис. 15.1. Вид фрейма на этапе конструирования (слева) и окно регистрации фрейма в палитре компонентов (справа)

7. После закрытия окна регистрации сам шаблон (см. рис. 15.1, слева) останется на экране. Подобно дополнительной форме его модуль становится составной частью проекта, поэтому размещение шаблона на форме не вызовет проблем.

Щелкните на значке  вкладки Standard палитры компонентов и затем — на пустом месте в форме Form1 — на экране появится окно с предложением выбрать нужный шаблон (рис. 15.2, сверху). Нажмите клавишу Enter, и шаблон появится в форме так, как если бы это был любой другой компонент из палитры компонентов (рис. 15.2, снизу).

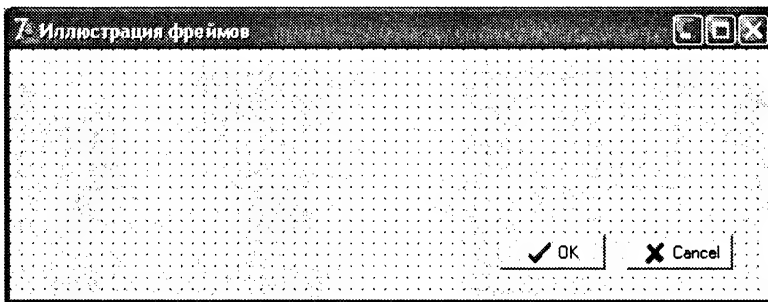
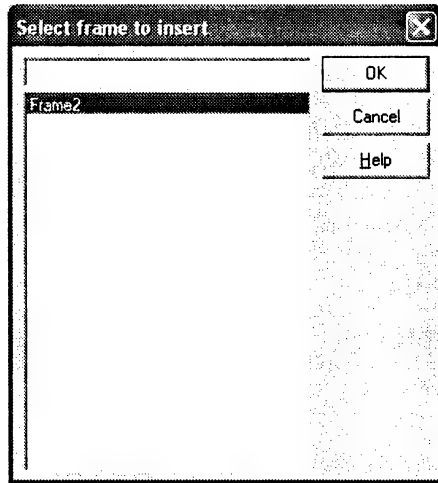


Рис. 15.2. Окно выбора доступного шаблона (сверху) и форма с размещенным на ней шаблоном (снизу)

Следует отметить, что, если вы откроете новый проект и попытаетесь разместить на форме фрейм с вкладки Standard, на экране вновь появится сообщение о том, что в проекте нет фреймов. Однако если вы присоедините модуль DlgBtnsF к проекту, нажав клавиши Shift+F11 или щелкнув на кнопке , установка шаблона на любую форму проекта приведет к появлению окна, показанного на рис. 15.2, сверху. Если в новом проекте попробовать установить на пустую форму компонент-шаблон с вкладки Templates (или с любой другой вкладки, куда он был помещен на этапе регистрации), на экране появится окно с таким сообщением (Модуль

DlgBtnsF необходим в вашем проекте, чтобы создать шаблон. Хотите его добавить?):

The following unit: DlgBtnsF, is needed in your project to create the template. Do you wish to add it?

TMainMenu — главное меню формы

Компонент класса TMainMenu определяет главное меню формы (программы). На форму можно поместить сколько угодно объектов этого класса, но отображаться в строке меню в верхней части формы будет только тот из них, который указан в свойстве Menu формы (это обстоятельство можно использовать для программного изменения меню формы).

После установки компонента на форму необходимо создать пункты меню. Для этого следует либо дважды щелкнуть на компоненте левой кнопкой мыши, либо щелкнуть на нем правой кнопкой и выбрать в контекстном меню команду Menu Designer, либо, наконец, щелкнуть в правой половине строки свойства Items в окне инспектора объектов. На рис. 15.3 показан вид окна конструктора меню, которое появится на экране после выполнения любого из перечисленных действий.

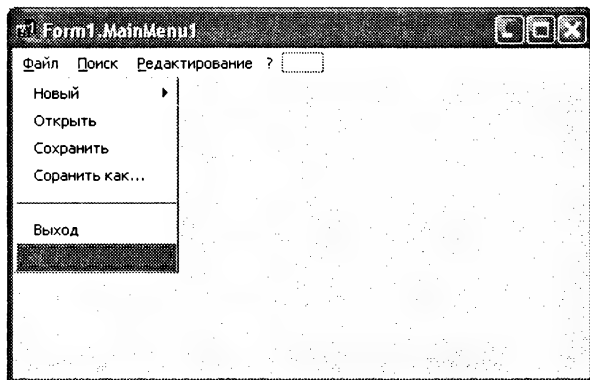


Рис. 15.3. Окно конструктора меню

Создание пунктов меню не вызывает проблем. Перейдите в окно инспектора объектов и введите название пункта меню в строке свойства Caption, после чего нажмите клавишу Enter — пункт меню готов и можно переходить к следующему. Каждый пункт главного меню может раскрываться в подменю или являться конечной командой. Для создания пунктов подменю щелкните мышью ниже пункта меню и введите имя первого пункта подменю. Продолжайте ввод, пока не будут созданы все пункты подменю, после чего щелкните на пустом прямоугольнике справа от первого пункта и введите имя следующего. Процесс гораздо сложнее описать, чем выполнить.

Чтобы связать с выбором пункта меню нужное действие, следует определить обработчик его события OnClick.

В названиях пунктов можно указать символ амперсанда (&) перед тем символом, который определит клавишу быстрого вызова (в терминологии Windows) этого

пункта. Например, на рис. 15.3 показано, что меню **Файл** можно раскрыть нажатием сочетания клавиш **Alt+Ф**. При создании меню этот пункт в строке свойства **Caption** инспектора объектов содержал текст **&Файл**.

Если вы захотите вставить разделительную черту, отделяющую пункты в подменю, введите в качестве имени очередного пункта меню дефис (-). Так, например, создана черта, отделяющая команды **Сохранить как** и **Выход** в меню **Файл** (см. рис.15.3): свойство **Caption** пятого по счету пункта меню **Файл** содержит единственный символ дефиса (-).

Для создания многоуровневых меню, то есть таких, у которых пункты подменю раскрывают новые подменю, щелкните на пункте подменю и нажмите клавиши **Ctrl+→**. Такого же эффекта можно добиться щелчком правой кнопки мыши на пункте подменю и выборе в контекстном меню команды **Create Submenu**. На рис. 15.4 показан пример многоуровневого меню.

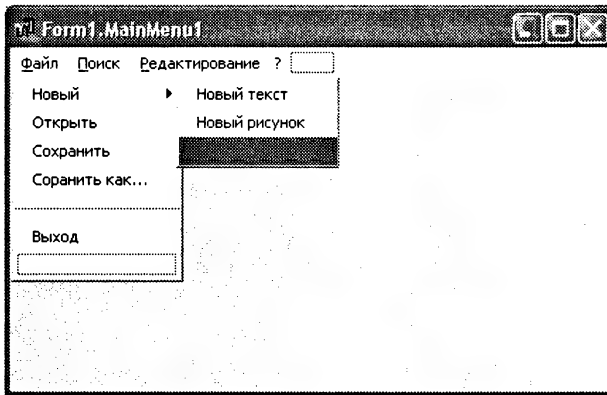


Рис. 15.4. Многоуровневое меню

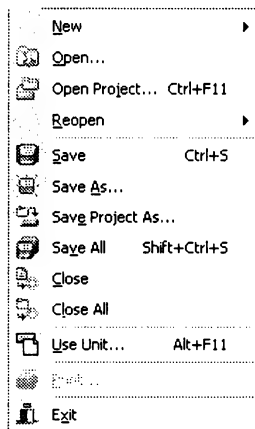


Рис. 15.5. Подменю File среды Delphi

В Delphi имеется возможность связывать с пунктами меню небольшие изображения. Эти изображения можно задать либо свойством **BitMap**, либо свойством

ImageIndex (этих свойств компонентов TMenu нет в ранних версиях Delphi). Изображение (если оно есть) появляется слева от пункта меню. Например, на рис. 15.5 показан фрагмент главного меню Delphi с раскрытым меню File. Как видно, команды Open, Open Project, Save и т. д. снабжены небольшими значками. Введение значков в меню, с одной стороны, повышает его наглядность, а с другой — способствует унификации значков в рамках механизма *действий* (см. раздел «Механизм действий» в главе 14). Если команда меню связана с каким-то действием своим свойством Action, а компонент TActionList, в котором это действие описано, в свою очередь, связан с хранилищем значков TImageList, индекс нужного значка можно задать в свойстве ImageIndex. В этом случае значок, указанный в свойстве BitMap (если он указан), игнорируется.

TRopupMenu — контекстное меню

Компоненты класса TRopupMenu используются для создания контекстных меню, появляющихся при щелчке на компоненте правой кнопкой мыши. В отличие от главного меню, контекстное меню может быть создано для *любого оконного компонента*. Чтобы связать щелчок правой кнопкой мыши на компоненте с раскрытием контекстного меню, в свойство RopupMenu компонента необходимо поместить имя компонента-меню.

Контекстное меню создается с помощью конструктора меню, процесс создания и свойства контекстного меню ничем не отличаются от создания меню класса TMainMenu.

TLabel — метка

Компоненты класса TLabel предназначены для размещения на форме различного рода текстовых надписей. Для этого служит центральное свойство компонента — Caption. С помощью свойства Font можно разнообразить вид надписи. Начиная с версии 2 с компонентом может быть связан оконный элемент управления, который выбирается при нажатии клавиш Alt+Буква, где Буква — выделенная подчеркиванием буква в тексте метки. Такие клавиши в терминологии Windows называются клавишами быстрого вызова.

С помощью логического свойства AutoSize можно разрешить компоненту автоматически изменять свои размеры так, чтобы полностью отобразить текст Caption с учетом установленного шрифта. Если свойство имеет значение False, высота и ширина компонента определяются текущими значениями свойств Height и Width. Если при этом ширина компонента больше минимально необходимой для отображения текста, свойством Alignment можно указать горизонтальное выравнивание текста: taLeftJustify — прижать текст влево; taCenter — выровнять по центру; taRightJustify — прижать текст вправо. Свойство Layout управляет положением текста по вертикали: tlTop — прижать вверх; tlCenter — выровнять по центру; tlBottom — прижать вниз.

Другие свойства метки перечислены в табл. 15.1.

Таблица 15.1. Свойства компонента TLabel

Свойство	Описание
property FocusControl: TWinControl;	Содержит имя оконного компонента, который связан с меткой клавишами быстрого вызова
property ShowAccelChar: Boolean;	Если содержит значение True, символ & в тексте метки предшествует символу клавиши быстрого вызова
property Transparent: Boolean;	Определяет прозрачность метки: если содержит значение False, пространство метки закрашивается собственным цветом Color, в противном случае этого не происходит, и сквозь метку будут видны расположенные за ней компоненты (так, например, можно наложить текст на графику)
property WordWrap: Boolean;	Разрешает/запрещает разрыв строки на границе слова. Для вывода многострочных надписей задайте условия AutoSize=False, WordWrap=True и установите подходящие размеры метки

ПРИМЕЧАНИЕ Метка Label может отображать длинную текстовую строку, указанную в свойстве Caption, в виде нескольких строк. Для этого установите в свойство AutoSize значение False, задайте достаточно большие размеры метки и поместите в свойство WordWrap значение True. Для отображения действительно многострочного текста используйте компоненты TMemo и TRichEdit.

TEdit — однострочное редактируемое текстовое поле

Компонент класса TEdit представляет собой однострочное редактируемое текстовое поле. С его помощью можно вводить и/или отображать достаточно длинные текстовые строки.

Центральным свойством компонента является свойство Text, которое содержит отображаемую компонентом строку. С помощью обработчика события OnChange программа может контролировать вводимый пользователем текст и при необходимости фильтровать его, игнорируя недопустимые символы. В следующем примере компонент фильтрует все символы, которые не соответствуют правильному представлению вещественного числа:

```
var
    OldText: String;
procedure TForm1.Edit1Change(Sender: TObject);
begin
    if Edit1.Text<>' ' then
        try
```

```

StrToFloat(Edit1.Text);
OldText := Edit1.Text
except
    // Ошибка преобразования: восстанавливаем прежний текст
    Edit1.Text := OldText;
    // и позиционируем текстовый курсор в конец текста:
    Edit1.SelStart := Length(Edit1.Text);
    Edit1.SelText := ''
end
end;

```

При воспроизведении примера подготовьте глобальную переменную `OldText` типа **String**, в которой запоминается последний правильно введенный текст. Лучше всего ее поместить в секцию **private** класса формы — тогда ее можно не обнулять в момент начала работы.

СОВЕТ Позиционировать текстовый курсор на любой символ строки можно с помощью свойств `SelStart` и `SelText`: в первое нужно поместить порядковый номер символа от начала текста, после которого должен стоять курсор ввода, во второе — пустую строку.

Свойства компонента `TEdit` представлены в табл. 15.2.

Таблица 15.2. Свойства компонента `TEdit`

Свойство	Описание
property <code>AutoSelect</code> : <code>Boolean</code> ;	Определяет, будет ли выделяться весь текст в момент получения компонентом фокуса ввода
property <code>AutoSize</code> : <code>Boolean</code> ;	Если содержит значение <code>True</code> и <code>BorderStyle = bsSingle</code> , высота компонента автоматически меняется при изменении свойства <code>Font.Size</code>
property <code>CanUndo</code> : <code>Boolean</code> ;	Содержит значение <code>True</code> , если сделанные пользователем изменения в тексте <code>Text</code> можно убрать методом <code>Undo</code>
<code>TEditCharCase = (ecNormal, ecUpperCase, ecLowerCase);</code> property <code>CharCase</code> : <code>TEditCharCase</code> ;	Определяет способ автоматического преобразования регистра букв: <code>ecNormal</code> — нет преобразования; <code>ecUpperCase</code> — все буквы прописные; <code>ecLowerCase</code> — все буквы строчные. Правильно работает с кириллицей
property <code>HideSelection</code> : <code>Boolean</code> ;	Если содержит значение <code>False</code> , выделение текста сохраняется при потере компонентом фокуса ввода
property <code>MaxLength</code> : <code>Integer</code> ;	Определяет максимальную длину текстовой строки. Если имеет значение 0, длина строки не ограничена
property <code>Modified</code> : <code>Boolean</code> ;	Содержит значение <code>True</code> , если текст был изменен
property <code>OnChange</code> : <code>TNotifyEvent</code> ;	Определяет обработчик события <code>OnChange</code> , которое возникает после любого изменения текста

Свойство	Описание
property OEMConvert: Boolean;	Содержит значение True, если необходимо перекодировать текст из кодировки MS-DOS в кодировку Windows и обратно
property PasswordChar: Char;	Если символ Char определен, он заменяет собой любой символ текста при отображении в окне. Используется для ввода паролей
property ReadOnly: Boolean;	Если содержит значение True, текст не может изменяться
property SelLength: Integer;	Содержит длину выделенной части текста
property SelStart: Integer;	Содержит номер первого символа выделенной части текста
property SelText: String;	Содержит выделенную часть текста
property Text: String;	Содержит весь текст

Замечу, что компонент TEdit, как и практически все остальные компоненты вкладки Standard, является «надстройкой» над стандартным объектом Windows. Поэтому, хотя его свойство Text объявлено как длинная строка **String**, максимальная емкость этого свойства лимитируется Windows и составляет 32 766 байт, в чем можно убедиться с помощью несложной программы (Chap_15\TEdit\Edit.dpr)¹.

Методы компонента TEdit перечислены в табл. 15.3.

Таблица 15.3. Методы компонента TEdit

Метод	Описание
procedure Clear;	Удаляет весь текст
procedure ClearSelection;	Удаляет выделенный текст
procedure ClearUndo;	Очищает буфер метода Undo
procedure CopyToClipboard;	Копирует выделенный текст в буфер обмена
procedure CutToClipboard;	Копирует выделенный текст в буфер обмена, после чего удаляет выделенный текст из компонента
function GetSelTextBuf(Buffer: PChar; BufSize: Integer): Integer;	Копирует не более BufSize символов выделенного текста в буфер Buffer
procedure PasteFromClipboard;	Заменяет выделенный текст содержимым буфера обмена, а если нет выделенного текста, копирует содержимое буфера обмена в позицию текстового курсора
procedure SelectAll;	Выделяет весь текст

— продолжение ➤

¹ Речь идет о Windows 98. Для Windows 2000/XP такого ограничения нет.

Таблица 15.3 (продолжение)

Метод	Описание
procedure SetSelTextBuf (Buffer: PChar);	Заменяет выделенный текст содержимым буфера Buffer, а если нет выделенного текста, копирует содержимое буфера Buffer в позицию текстового курсора
procedure Undo;	Восстанавливает текст в том виде, в котором он был перед последним получением компонентом фокуса ввода

ТМемо — многострочное редактируемое текстовое поле

Компоненты класса ТМемо предназначены для ввода, редактирования и/или отображения достаточно длинного многострочного текста. Текст хранится в свойстве Lines класса TStrings и, таким образом, представляет собой пронумерованный набор строк (нумерация начинается с нуля). С помощью свойств и методов этого класса (Count, Add, Delete, Clear и т. д. — см. раздел «Классы TStrings и TStringList — наборы строк и объектов» в главе 13) можно динамически формировать содержимое компонента.

Свойства BorderStyle, CanUndo, HideSelection, MaxLentgh, Modified, OEMConvert, OnChange, ReadOnly, SelLength, SelStart и SelText аналогичны соответствующим свойствам класса TEdit. Свойство WordWrap аналогично свойству TLabel.WordWrap. Другие специфические свойства представлены в табл. 15.4.

Таблица 15.4. Свойства компонента ТМемо

Свойство	Описание
property CaretPos: TPoint;	Содержит координаты мигающего текстового курсора относительно границ клиентской области компонента
property Lines: TStrings;	Содержит строки текста
TScrollStyle = (ssNone, ssHorizontal, ssVertical, ssBoth); property ScrollBars: TScrollStyle;	Определяет наличие в многострочном поле полос прокрутки: ssNone — нет полос; ssHorizontal — есть горизонтальная полоса; ssVertical — есть вертикальная полоса; ssBoth — есть обе полосы
property Text: String;	Отображает содержимое свойства Lines в виде одной длинной строки, в которой границы отдельных строк определяются символами EOL (CR+LF)
property WantReturns: Boolean;	Если содержит значение True, нажатие клавиши Enter вызывает переход на новую строку, в противном случае — обрабатывается системой. Для перехода на новую строку в этом случае следует нажать клавиши Ctrl+Enter

Свойство	Описание
property WantTabs: Boolean;	Если содержит значение True, нажатие клавиши Tab вызывает ввод в текст символа табуляции, в противном случае — обрабатывается системой. Для ввода символа табуляции в этом случае следует нажать клавиши Ctrl+Tab

Замечу, что, если свойство `ScrollBars` содержит значение `ssHorizontal` или `ssBoth`, свойство `WordWrap` игнорируется, и длинные строки будут отсекаться границами компонента без переноса текста на следующую строку.

Специфические методы класса аналогичны методам класса `TEdit`.

Поскольку компонент является потомком класса `TControl`, он имеет также свойство `Text`, которое содержит отображаемый компонентом текст в виде одной длинной строки. В этой цепочке символов границы строк многострочного текста выделяются символами с кодом `#13#10` (признак `EOLN` — конец строки). В отличие от этого свойство `Lines` содержит пронумерованный список строк: первая строка в этом списке имеет индекс 0, вторая — 1, а общее количество строк можно узнать с помощью метода `Lines.Count`. Свойство `Text` удобно использовать для поиска в тексте нужного фрагмента. Чтобы, например, найти и выделить в тексте фрагмент, содержащийся в компоненте `edSearch` типа `TEdit`, можно использовать такой обработчик события `OnClick` кнопки `btSearch`:

```
procedure TForm1.btSearchClick(Sender: TObject);  
var  
    k: Integer;  
begin  
    with Mem01 do  
        begin  
            k := pos(edSearch.Text, Text);  
            if k>0 then  
                begin  
                    SelStart := k-1;  
                    SelLength := Length(edSearch.Text)  
                end  
            end;  
        end;  
end;
```

Если вы хотите, чтобы найденный в тексте фрагмент после установки свойств `SelStart` и `SelLength` сразу оказался выделенным, установите значение `False` в свойство `HideSelection`.

Как и в компоненте `TEdit`, максимальная длина свойства `Text` ограничивается `Windows` и составляет 40 976 байт (проект `Chap_15\ТMemo\Memo.dpr`)¹.

Для загрузки в компонент текста из файла и для сохранения текста в файле удобно использовать методы `LoadFromFile` и `SaveToFile` класса `TStrings`. Например, следующий обработчик события `OnCreate` формы `Form1` загружает в компонент `Mem01` текст проектного файла программы:

¹ Речь идет о `Windows 98`. Для `Windows 2000/XP` такого ограничения нет.


```

procedure TForm1.FormCreate(Sender: TObject);
begin
    Memo1.Lines.LoadFromFile(
        ChangeFileExt(Application.ExeName, '.dpr'))
end;

```

TButton — кнопка

Компоненты TButton широко используются для управления программами. Связанный с кнопкой алгоритм управления реализуется в обработчике события OnClick.

Свойства компонента представлены в табл. 15.5.

Таблица 15.5. Свойства компонента TButton

Свойство	Описание
property Cancel: Boolean;	Если имеет значение True, событие OnClick кнопки возникает при нажатии клавиши Esc
property Default: Boolean;	Если имеет значение True, событие OnClick кнопки возникает при нажатии клавиши Enter
type TModalResult = Low(Integer)..High(Integer); property ModalResult: TModalResult;	Определяет результат, с которым было закрыто модальное окно (см. ниже пояснение)

В терминологии Windows *модальными окнами* называются такие специальные окна, которые, появившись на экране, блокируют работу пользователя с другими окнами вплоть до своего закрытия. Обычно с их помощью реализуется диалог, требующий от пользователя принятия некоторого решения. Для этого в состав модального окна включается несколько кнопок. Если у кнопки определено свойство ModalResult, щелчок на ней приводит к закрытию модального окна и возвращению в программу значения ModalResult как результата диалога с пользователем. В Delphi определены следующие стандартные значения ModalResult:

- mrNone — модальное окно не закрывается;
- mrOk — была нажата кнопка OK;
- mrCancel — была нажата кнопка Cancel;
- mrAbort — была нажата кнопка Abort;
- mrRetry — была нажата кнопка Retry;
- mrIgnore — была нажата кнопка Ignore;
- mrYes — была нажата кнопка Yes;
- mrNo — была нажата кнопка No;
- mrAll — была нажата кнопка All.

В отличие от большинства других видимых компонентов кнопка TButton является элементом операционной системы Windows и поэтому не может изменять

свой цвет произвольным образом — кнопка меняет его вместе с изменением палитры Windows. Кнопка всегда имеет системный цвет `clBtnFace` и не имеет свойства `Color`. Шрифт надписи на кнопке может менять свой стиль и размер, но компонент игнорирует изменение его цвета.

TCheckBox — флажок

Компонент `TCheckBox` используется для того, чтобы пользователь мог указать свое решение типа *да/нет* или *да/нет/не уверен* (в последнем случае флажок устанавливается, но выглядит блеклым). Это решение отражается в свойстве `State` компонента, доступном как для чтения, так и для записи. В составе диалогового окна может быть несколько компонентов `TCheckBox`, причем состояние любого из них никак не зависит от состояния остальных.

Типичное использование компонента:

```
if CheckBox1.Checked then
    ...
else
    ...
```

Или:

```
case CheckBox1.State of
  cbChecked : ...;
  cbUnchecked: ...;
  cbGrayed : ...;
end;
```

Свойства компонента `TCheckBox` представлены в табл. 15.6.

Таблица 15.6. Свойства компонента `TCheckBox`

Свойство	Описание
type <code>TLeftRight = (taLeftJustify, taRightJustify);</code> property <code>Alignment: TLeftRight;</code>	Определяет положение текста: <code>taLeftJustify</code> — с левой стороны компонента; <code>taRightJustify</code> — с правой стороны
property <code>AllowGrayed: Boolean;</code>	Разрешает/запрещает использование состояния <code>cbGrayed</code> (не уверен)
property <code>Caption: String;</code>	Содержит связанный с компонентом текст
property <code>Checked: Boolean;</code>	Содержит значение, соответствующее выбору пользователя. Состоянию <code>cbChecked</code> соответствует значение <code>True</code> . Состояния <code>cbUnchecked</code> и <code>cbGrayed</code> отражаются как <code>False</code>
type <code>TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed);</code> property <code>State: TCheckBoxState;</code>	Содержит состояние компонента: <code>cbUnchecked</code> — не установлен; <code>cbChecked</code> — установлен; <code>cbGrayed</code> — не уверен

Свойство `Color` компонента в Windows 98 игнорируется. Свойства `Height` и `Width` определяют размеры прямоугольника, в котором выводится связанный с флажком текст, и никак не влияют на размеры самого флажка. Текст указывается в свойстве `Caption`.

TRadioButton — переключатель

В отличие от компонента `TCheckBox` компоненты `TRadioButton` представляют собой переключатели, предназначенные для выбора одного из нескольких взаимоисключающих решений. На форму (точнее, в компонент-контейнер) помещается, по меньшей мере, два таких компонента. Они могут иметь только два состояния, определяемых свойством `Checked`. Если в одном компоненте это свойство принимает значение `True`, во всех других компонентах, расположенных в том же контейнере, свойства `Checked` принимают значения `False`.

Помимо свойства `Checked` компонент `TRadioButton` имеет еще одно специфичное свойство — `Alignment`, аналогичное такому же свойству компонента `TCheckBox`. Как и в случае с флажком, программист может изменять размеры и цвет текста, указанного в свойстве `Caption`, но не может изменять размеры и цвет самого переключателя (в Windows 98).

TListBox — список

Компонент класса `TListBox` представляет собой стандартный для Windows список, с помощью которого пользователь может выбрать один или несколько пунктов. В компоненте предусмотрена возможность программной прорисовки пунктов, поэтому список может содержать не только текстовые строки, но и произвольные изображения.

Свойства компонента `TListBox` перечислены в табл. 15.7.

Таблица 15.7. Свойства компонента `TListBox`

Свойство	Описание
property <code>AutoComplete:</code> <code>Boolean;</code>	Если содержит значение <code>True</code> , выбранный элемент в списке замаскируется тем, начальные буквы которого вводит пользователь
type <code>TborderStyle =</code> <code>bsNone.. bsSingle;</code> property <code>BorderStyle:</code> <code>TborderStyle;</code>	Определяет стиль рамки: <code>bsNone</code> — нет рамки; <code>bsSingle</code> — рамка толщиной 1 пиксел
property <code>Canvas:</code> <code>TCanvas;</code>	Канва для программной прорисовки пунктов списка
property <code>Columns:</code> <code>LongInt;</code>	Определяет количество колонок в списке
property <code>ExtendedSelect:</code> <code>Boolean;</code>	Если <code>ExtendedSelect=True</code> и <code>MultiSelect=True</code> , выбор пункта без одновременного нажатия клавиши <code>Ctrl</code> или <code>Alt</code> отменяет предыдущий выбор
property <code>IntegralHeight:</code> <code>Boolean;</code>	Если <code>IntegralHeight=True</code> и <code>Style<>lbOwnerDrawVariable</code> , в списке отображается целое количество пунктов

Свойство	Описание
property ItemHeight: Integer;	Определяет высоту элемента в пикселах для Style=lbOwnerDrawFixed
property ItemIndex: Integer;	Содержит индекс элемента, имеющего фокус ввода. Если MultiSelect=False, совпадает с индексом выделенного пункта
property Items: TStrings;	Содержит набор строк, показываемых в компоненте
property MultiSelect: Boolean;	Разрешает/отменяет возможность выбора нескольких пунктов
property SelCount: Integer;	Содержит количество выбранных пунктов
property Selected[X: Integer]: Boolean;	Содержит признак выбора для элемента с индексом X (первый элемент имеет индекс 0)
property Sorted: Boolean;	Разрешает/отменяет сортировку строк в алфавитном порядке
type TListBoxStyle = (lbStandard, lbOwnerDrawFixed, lbOwnerDrawVariable); property Style: TListBoxStyle;	Определяет способ прорисовки элементов: lbStandard — элементы рисует Windows; lbOwnerDrawFixed — элементы рисует программа, и они имеют одинаковую высоту, определяемую свойством ItemHeight; lbOwnerDrawVariable — элементы рисует программа, и они имеют разную высоту
property TabWidth: Integer;	Задаёт ширину табуляционного пробела
property TopIndex: Integer;	Индекс первого видимого в окне пункта списка

Создание элементов (пунктов) списка компонента реализуется с помощью методов его свойства Items — Add, Append, Insert или LoadFromFile.

Следует учесть, что свойство ItemIndex в компонентах-списках TListBox и TComboBox (см. ниже) содержит индекс элемента списка, имеющего фокус ввода, и не меняется при изменении списка методами Add, Append, Insert. Следующий цикл, в котором каждому элементу списка Items ставится в соответствие свой список TStringList (Items является специализированным потомком класса TStrings и, следовательно, может содержать не только строки, но и объекты), ошибочен:

```

while (...) do
begin
    Items.Add('...');
    Objects[Index] := TStringList.Create;
    ...
end;
```

Так как при вставке очередной строки значение ItemIndex не меняется, объект и связанный с ним список TStringList будет создан только для одной строки компонента. Исправленный вариант:

```

k := 0;
while (...) do
begin
  Items.Add('...');
  Objects[k] := TStringList.Create;
  inc(k);
  ...
end;

```

Учтите также, что значение свойства `ItemIndex` не может быть больше *или равно* значению `Items.Count`, поэтому, если нет искомого элемента списка, такой цикл будет бесконечным:

```

while (ItemIndex < Items.Count)
    ...
    // Ошибка! ItemIndex не может
    // быть равен Items.Count
and (Items[ItemIndex] <> ...) do
  ItemIndex := ItemIndex+1;

```

Нужно исключить переход в цикле за границы списка:

```

while (ItemIndex < Items.Count-1) // Теперь все правильно
  and (Items[ItemIndex] <> ...) do
  ItemIndex := ItemIndex+1;
if Items[ItemIndex] <> ...
    // Однако приходится дополнительно
then ItemIndex := -1; // проверять последнюю строку

```

Для компонента определены два события, связанные с программной прорисовкой элементов списка:

```

type
  TOwnerDrawState = set of (odSelected, odGrayed, odDisabled,
                             odChecked, odFocused);
  TDrawItemEvent = procedure(Control: TWinControl;
                             Index: Integer; Rect: TRect;
                             State: TOwnerDrawState) of
object;
property OnDrawItem: TDrawItemEvent;
type
  TMeasureItemEvent = procedure(Control: TWinControl;
                                Integer; var Height: Integer) of object;
property OnMeasureItem: TMeasureItemEvent;

```

Первое событие возникает в момент, когда программа должна нарисовать очередной элемент. Обработчик события получает ссылку на список `Control`, индекс изображаемого элемента `Index`, границы элемента `Rect` и его состояние `State`. Прорисовка ведется с помощью свойства `Canvas`.

ПРИМЕЧАНИЕ

Поскольку программная прорисовка обычно связана с выводом на экран рисунков, параметр `Control` задан как абстрактный объект класса `TWinControl`, и необходимое преобразование типов должно выполняться в обработчике.

В следующем примере в каждом элементе рисуется растровое изображение и текст.

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl;
    Index: Integer; Rect: TRect; State:
    TOwnerDrawState);

var
    Bitmap: TBitmap; // Временная переменная для растра
    Offset: Integer; // Расстояние от растра до текста
begin
    { Свойство Canvas имеет не только список ListBox, но и
    форма, на которую он помещен! Поэтому нужно явно указать
    канву. Преобразуем тип и указываем канву: }
    with Control as TListBox, Canvas do
        begin
            FillRect(Rect); // Очищаем прямоугольник
            // Получаем из списка растр:
            Bitmap := TBitmap(Items.Objects[Index]);
            if Bitmap <> nil then
                begin
                    // Вычерчиваем растр:
                    BrushCopy(Bounds(Rect.Left + 2, Rect.Top,
                        Bitmap.Width,
                        Bitmap.Height), Bitmap,
                        Bounds(0, 0, Bitmap.Width,
                        Bitmap.Height), clRed);
                    // Вычисляем смещение текста (4 пиксела от растра):
                    Offset := Bitmap.width + 6;
                end
                else Offset := 2; // Если нет растра, 2 пиксела слева
                // Выводим текст:
                TextOut(Rect.Left + Offset, Rect.Top, Items[Index]);

            end;
        end;
```

Обратите внимание: обработчик не учитывает состояние элемента. Окантовка сфокусированного элемента пунктиром, установка нужного цвета кисти и шрифта для выбранного элемента осуществляются автоматически. Если вас не устраивают стандартные цвета, следует проверять параметр State. Например, в предыдущем примере обратите внимание на следующий оператор:

```
FillRect(Rect); // Очищаем прямоугольник
```

Если перед этим оператором вставить показанные ниже строки, выбранные элементы будут подсвечены желтым фоном, а текст в них будет выведен черным полужирным шрифтом:

```
if odSelected in State then
    begin
```

```

    Brush.Color := clYellow;
    Font.Color := clBlack;
    Font.Style := [fsBold];
end;

```

Событие `OnMeasureItem` возникает только при выполнении условия `Style=lbOwnerDrawVariable`. Оно предшествует событию `OnDrawItem`, и в ходе его обработки программа должна установить нужную высоту очередного элемента. Обработчику передается индекс элемента `Index` и переменная `Height`, в которой он должен вернуть высоту элемента. Например:

```

procedure TForm1.ListBox1MeasureItem(Control: TWinControl;
    Index: Integer; var Height: Integer);
var
    Bitmap: TBitmap;
begin
    Bitmap := TBitmap((Control as
        TListBox).Items.Objects[Index]);
    if Bitmap <> nil then
        Height := Bitmap.Height
    else
        Height := abs(ListBox1.Font.Height)
    end;

```

TComboBox — комбинированный список

Комбинированный, или раскрывающийся, список `TComboBox` представляет собой комбинацию списка `TListBox` и текстового поля `TEdit`, и поэтому большая часть его свойств и методов заимствованы у этих компонентов. Существуют пять модификаций компонента, определяемые его свойством `Style`: `csSimple`, `csDropDown`, `csDropDownList`, `csOwnerDrawFixed` и `csOwnerDrawVariable`. В первом случае список всегда закрыт. В модификации `csSimple` для смены элемента списка, демонстрируемого в его окне, следует активизировать компонент и использовать клавиши смещения курсора $\downarrow$ и $\uparrow$. Программист не может раскрыть список, даже если пошлет ему сообщение `sc_ShowDropDown`, предусмотренное в Windows специально для раскрытия списка: это сообщение действительно только для остальных модификаций, в которых он раскрывается после щелчка на кнопке справа от поля списка (рис. 15.6). В модификации `csDropDownList` поле списка работает в режиме отображения выбранного элемента, и в это поле нельзя ввести новый элемент (в других модификациях это возможно). Модификации `csOwnerDrawFixed` и `csOwnerDrawVariable` используются для программной прорисовки элементов списка. Требуемые для этого свойства и методы полностью совпадают со свойствами и методами компонента `TListBox` аналогичного назначения.

Фактически «своими» у компонента являются лишь свойства и события, связанные с раскрытием списка:

```

property DropDownCount: Integer;
property DroppedDown: Boolean;
property OnDropDown: TNotifyEvent;

```

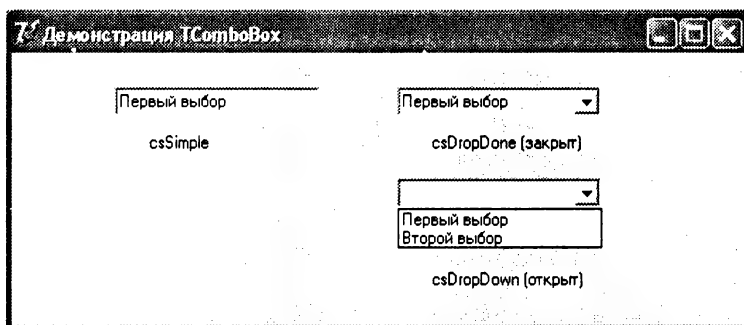


Рис. 15.6. Примеры комбинированных списков

Свойство `DropDownCount` определяет количество элементов списка, появление которых еще не приводит к необходимости прокрутки списка. По умолчанию это свойство имеет значение 8: если в списке указано 9 и более элементов (то есть больше, чем содержит свойство `DropDownCount`), при его раскрытии к окну будет добавлена полоса прокрутки. Свойство `DroppedDown` определяет, раскрыт ли в данный момент список. Это свойство доступно также для записи, что позволяет программно управлять состоянием списка. Событие `OnDropDown` происходит при изменении состояния списка.

Наполнение списка ведется методами `Add`, `Append`, `Insert` и т. п. его свойства `Items` класса `TStrings` (см. раздел «Классы `TStrings` и `TStringList` — наборы строк и объектов» в главе 13).

TScrollBar — полоса прокрутки

С помощью компонента `TScrollBar` пользователи создают ползунки — стандартные для Windows элементы управления, похожие на полосы прокрутки окна. Обычно такие элементы управления используются для плавного изменения числовой величины (см. также раздел «`TTrackBar` — ползунок» в главе 17).

Свойства компонента `TScrollBar` представлены в табл. 15.8.

Таблица 15.8. Свойства компонента `TScrollBar`

Свойство	Описание
<code>TScrollBarKind = (sbHorizontal, sbVertical);</code> property <code>Kind: TScrollBarKind;</code>	Определяет ориентацию компонента: <code>sbHorizontal</code> — бегунок перемещается по горизонтали; <code>sbVertical</code> — бегунок перемещается по вертикали
property <code>LargeChange: TScrollBarInc;</code>	«Большой» сдвиг бегунка (при щелчке мышью рядом с концевой кнопкой)
property <code>Max: Integer;</code>	Максимальное значение диапазона изменения числовой величины
property <code>Min: Integer;</code>	Минимальное значение диапазона изменения числовой величины

продолжение ➤

Таблица 15.8 (продолжение)

Свойство	Описание
property Position: Integer;	Текущее значение числовой величины
property SmallChange: TScrollBarInc;	«Малый» сдвиг бегунка (при щелчке мышью на концевой кнопке)

С помощью следующего метода можно сразу установить свойства Position, Max и Min:

```
procedure SetParams(APosition, AMax, AMin: Integer);
```

С компонентом связано два события:

```
type TScrollEvent = procedure (Sender: TObject;  
    ScrollCode: TScrollCode; var ScrollPos: Integer) of  
    object;  
property OnScroll: TScrollEvent;  
property OnChange: TNotifyEvent;
```

Первое возникает при любом изменении свойства Position, второе — при изменении параметров методом SetParams.

Подобно компоненту TButton компонентом TScrollBar полностью управляет Windows, поэтому у него нет свойства Color.

TGroupBox — панель группирования

Этот компонент служит контейнером для размещения дочерних компонентов и представляет собой прямоугольную рамку и текст в разрыве рамки. Обычно с его помощью выделяется группа управляющих элементов, объединенных по функциональному назначению. Свойства и методы этого класса целиком унаследованы им от своих предков TCustomControl и TWinControl и описаны в главе 14.

TRadioGroup — группа переключателей

Компонент класса TRadioGroup представляет собой специальный контейнер, предназначенный для размещения переключателей класса TRadioButton. Каждый размещаемый в нем переключатель помещается в специальный список Items и доступен по индексу, что упрощает обслуживание группы.

Свойства компонента TRadioGroup перечислены в табл. 15.9.

Таблица 15.9. Свойства компонента TRadioGroup

Свойство	Описание
property Columns: Integer;	Определяет количество столбцов переключателей
property ItemIndex: Integer;	Содержит индекс установленного переключателя

Свойство	Описание
property Items: TStrings;	Содержит список строк с заголовками элементов. Добавление/удаление элементов достигается добавлением/удалением строк списка Items

После размещения компонента на форме он пуст. Чтобы создать в нем хотя бы один переключатель, следует раскрыть редактор списка Items и ввести хотя бы одну строку: строки Items используются как поясняющие надписи справа от переключателей, а их количество определяет количество переключателей в группе. Замечу также, что после создания компонента его свойство `ItemIndex` по умолчанию имеет значение `-1`; это означает, что ни один переключатель в группе не установлен. Чтобы в момент появления компонента на экране какой-то переключатель был уже установлен, необходимо на этапе конструирования с помощью окна инспектора объектов или программно (например, в обработчике `OnActivate` формы) установить в свойство `ItemIndex` номер соответствующего переключателя (нумерация начинается с 0). Это же свойство позволяет программе проанализировать выбор пользователя, например:

```

case RadioGroup1.ItemIndex of
  0: ...;  // Установлен 1-й переключатель
  1: ...;  // Установлен 2-й переключатель
  ...
else
  ...      // Не установлен ни один из переключателей
end;

```

TPanel — панель

Компонент `TPanel` представляет собой контейнер общего назначения. В отличие от компонента `TGroupBox` он не имеет заголовка и поэтому менее удобен для функционального группирования элементов. С другой стороны, его свойство `Caption` отображается в виде текстовой строки и может использоваться для вывода сообщений. Компоненты этого класса часто помещаются на форму для того, чтобы располагать вставленные в них дочерние компоненты вдоль одной из сторон окна независимо от изменения размеров этого окна.

Компонент имеет развитые средства создания различных эффектов объемности за счет использующихся в нем двух кромок — внешней и внутренней.

Свойства компонента `TPanel` представлены в табл. 15.10.

Таблица 15.10. Свойства компонента `TPanel`

Свойство	Описание
<code>TBorderStyle = bsNone..bsSingle;</code> property BorderStyle: <code>TBorderStyle;</code>	Определяет стиль рамки: <code>bsNone</code> — нет рамки; <code>bsSingle</code> — компонент по периметру обводится линией толщиной в 1 пиксел
property FullRepaint: <code>Boolean;</code>	Разрешает/запрещает перерисовку панели и всех ее дочерних элементов при изменении ее размеров

Для компонента объявлено событие `OnResize`, в обработчике которого программист может предусмотреть необходимую реакцию на изменение размеров компонента.

TActionList — список действий

Этот компонент не имеет видимого изображения и служит для поддержки механизма действий (см. раздел «Механизм действий» в главе 14).

Основная схема его использования такова. Вначале с помощью его редактора создается *действие* — объект класса `TAction`. Редактор (рис. 15.7) вызывается двойным щелчком на компоненте или выбором команды `Action List Editor` в его контекстном меню, которое раскрывается после щелчка на нем правой кнопкой мыши.

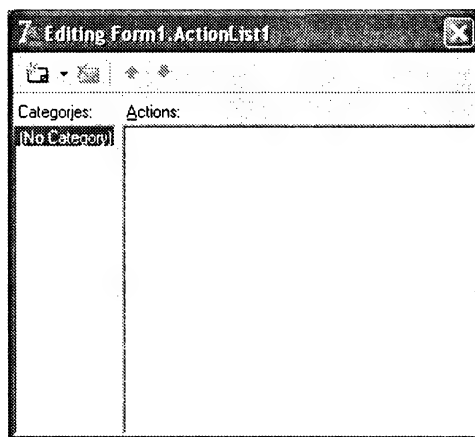


Рис. 15.7. Редактор компонента `TActionList`

Этот объект имеет ряд свойств и событий, с помощью которых уточняется характер действия. Доступ к этим свойствам и событиям можно получить из окна инспектора объектов. С действием можно связать группу свойств (`Caption`, `Checked`, `Enabled`, `ShortCut` и т. д.), которые будут помещаться в одноименные свойства компонентов, реализующих общее действие. Если с компонентом связан контейнер значков типа `TImageList` (свойство `Images` — это свойство не действия, а свойства самого компонента `TActionList`), при реализации действия можно использовать один из хранящихся в нем значков (`ImageIndex`). Чтобы действие подкреплялось программным кодом, для него обязательно следует определить обработчик события `OnExecute`.

В свойства `Action` тех компонентов, активизация которых должна сопровождаться одним и тем же действием (например, в свойства `Action` команды меню и кнопки панели инструментов), помещается имя только что определенного действия. В результате выбор команды в меню или щелчок на кнопке панели инструментов вызовут один и тот же метод (`OnExecute`). Кроме того, возле команды меню и на кнопке панели инструментов будет присутствовать один и тот же зна-

чок (ImageIndex), у них будут одинаковые названия (Caption), оперативные подсказки (Hint) и т. д.

Свойства компонента TActionList представлены в табл. 15.11.

Таблица 15.11. Свойства компонента TActionList

Свойство	Описание
property ActionCount: Integer;	Содержит количество определенных в компоненте действий (только для чтения)
property Actions[Index: Integer]: TContainedAction;	Позволяет программе обратиться к нужному действию (объекту класса TContainedAction) по его индексу Index
property Images: TCustomImageList;	Содержит имя компонента класса TimageList

Редактор компонента создает объекты класса TAction. Свойства класса TAction, в основном, определяют те значения, которые будут иметь одноименные свойства всех компонентов или команд меню, связанных общим действием (если, разумеется, компонент или команда меню имеет данное свойство).

Свойства класса TAction перечислены в табл. 15.12.

Таблица 15.12. Свойства класса TAction

Свойство	Описание
property Caption: String;	Содержит строку, которая будет устанавливаться в свойствах Caption всех компонентов, связанных данным действием
property Checked: Boolean;	Содержит значение, которое будет устанавливаться в свойствах Checked
property DisableIfNoHandler: Boolean;	Указывает, будут ли недоступны связанные компоненты, если для действия не определен обработчик OnExecute
property Enabled: Boolean;	Содержит значение, которое будет устанавливаться в свойствах Enabled
property HelpContext: THelpContext;	Содержит значение, которое будет устанавливаться в свойствах HelpContext
property Hint: String;	Содержит строку, которая будет устанавливаться в свойствах Hint
property ImageIndex: Integer;	Содержит индекс изображения в хранилище, указанном в свойстве Images компонента. Это изображение будет связано с компонентами данного действия
property ShortCut: TShortCut;	Содержит значение, которое будет устанавливаться в свойствах ShortCut
property Visible: Boolean;	Содержит значение, которое будет устанавливаться в свойствах Visible

События класса TAction перечислены в табл. 15.13.

Таблица 15.13. События класса TAction

Событие	Описание
property OnExecute: TNotifyEvent;	Возникает при щелчке мышью на одном из компонентов, связанных общим действием. Обработчик этого события должен реализовать нужное действие
THintEvent = procedure (var HintStr: String ; var CanShow: Boolean) of object ; property OnHint: THintEvent;	Возникает при помещении указателя мыши на одном из связанных общим действием компонентов. Его обработчик по умолчанию выводит оперативную подсказку со строкой HintStr
property OnUpdate: TNotifyEvent;	Возникает, когда очередь сообщений для приложения пуста или когда обновляется содержимое списка действий

Глава 16

Компоненты вкладки Additional

На вкладке Additional расположены компоненты общего назначения. Некоторые из них в функциональном плане дублируют компоненты вкладки Standard (см. главу 15), но большая часть имеют уникальную функциональность.

TBitBtn — кнопка с изображением

Графическая кнопка TBitBtn представляет собой популярную разновидность стандартной кнопки TButton. Ее отличительная особенность — свойство Glyph, с помощью которого определяется растровое изображение на поверхности кнопки. В комплект Delphi входит множество рисунков, разработанных специально для размещения на этих кнопках. Так как дальше мне придется еще не раз использовать поставляемые с Delphi изображения, то следующий текст я выделяю, и в дальнейшем буду просто ссылаться на показанное ниже примечание.

ПРИМЕЧАНИЕ

По умолчанию для версий Delphi 4–7 рисунки для кнопок размещаются в папке Program Files\Common Files\Borland Shared\Images\Buttons, для остальных версий — в папке Images\Buttons каталога размещения Delphi.

Свойство Kind определяет одну из 11 стандартных разновидностей кнопки, показанных на рис. 16.1.

Щелчок на любой из показанных на рисунке кнопок, кроме кнопок bkCustom и bkHelp, закрывает модальное окно и возвращает в программу результат вида mrXXX (bkOk — mrOk, bkCancel — mrCancel и т. д.). Кнопка bkClose для модального окна возвращает значение mrCancel, а для главного окна программы — закрывает его и завершает работу программы. Кнопка bkHelp автоматически вызывает раздел справочной службы, связанный со свойством HelpContext формы, на которую кнопка помещена. Если у кнопки был изменен рисунок (свойство Glyph), Delphi автоматически присвоит свойству Kind кнопки значение bkCustom.



Рис. 16.1. Разновидности кнопок TBitBtn

Это произойдет также в случае, когда кнопка указана как кнопка по умолчанию (Default=True), но при этом свойство Kind не содержит значение bkOK или bkYes, и, кроме того, когда ее свойство Cancel содержит значение True, а свойство Kind не содержит значение bkCancel или bkNo.

Свойства Cancel, Default и ModalResult кнопки TBitBtn унаследовала у своего родительского класса TButton, остальные специфические свойства перечислены в табл. 16.1.

Таблица 16.1. Свойства компонента TBitBtn

Свойство	Описание
property Glyph: TBitmap;	Определяет от 1 до 4 связанных с кнопкой растровых изображений (см. ниже)
TBitBtnKind = (bkCustom, bkOK, bkCancel, bkHelp, bkYes, bkNo, bkClose, bkAbort, bkRetry, bkIgnore, bkAll); property Kind: TBitBtnKind;	Определяет разновидность кнопки
TButtonLayout = (blGlyphLeft, blGlyphRight, blGlyphTop, blGlyphBottom); property Layout: TButtonLayout;	Определяет край кнопки, к которому прижимается рисунок: blGlyphLeft — к левому; blGlyphRight — к правому; blGlyphTop — к верхнему; blGlyphBottom — к нижнему
property Margin: Integer;	Определяет расстояние в пикселах от края кнопки до рисунка
TNumGlyphs: 1..4; property NumGlyphs: TNumGlyphs;	Определяет количество растровых изображений (см. ниже)
property Spacing: Integer;	Определяет расстояние в пикселах от рисунка до надписи на кнопке
TButtonStyle = (bsAutoDetect, bsWin31, bsNew);	Определяет стиль оформления кнопки, зависящий от операционной системы. Стиль bsNew соответствует 32-разрядным версиям Windows.

Свойство	Описание
property Style: TButtonStyle;	Стиль bsAutoDetect изменяет оформление кнопки в зависимости от операционной системы, под управлением которой работает программа в данный момент

При разработке собственных растровых изображений для использования на кнопках следует учесть, что изображения должны меняться при изменении состояния кнопки. Таких состояний может быть четыре: нормальное состояние, кнопка недоступна, кнопка нажата и кнопка утоплена (последнее используется только в кнопках TSpeedButton). В соответствии с этим разрабатывается до 4 изображений, расположенных по горизонтали в виде одного длинного растра. Например, стандартный размер изображения для размещения на кнопке равен 16×16 пикселей. Если создаются 3 изображения, размер растра должен составлять 16×48. Количество изображений в растре задается свойством NumGlyphs. Если каждое изображение квадратное, и длина растра делится без остатка на его высоту, Delphi автоматически распознает количество изображений. Если задано только одно изображение, оно меняется автоматически: в состоянии «кнопка нажата» изображение смещается на один пиксел вправо и вниз, а в состоянии «кнопка недоступна» все цвета, кроме черного, меняются на светло-серый, а черный — на белый, что обеспечивает эффект «вдавленности» изображения. Следует также учесть, что самый левый нижний пиксел растра определяет цвет прозрачности: на кнопке этот цвет будет заменяться цветом поверхности кнопки.

Как и в кнопках TButton, программист не может управлять цветом поверхности кнопки, но в отличие от TButton может менять цвет надписи на ней. С помощью свойства Default кнопку можно сделать кнопкой, выбранной по умолчанию, — в этом случае нажатие клавиши Enter автоматически вызывает обработчик ее события OnClick. Однако кнопками, выбранными по умолчанию, можно сделать только кнопки bkYes и bkNo, остальные значения свойства Kind будут заменяться на bkCustom при вводе в свойство Default значения True, и наоборот — в свойство Default автоматически помещается значение False при установке в свойстве Kind любого значения, кроме bkYes, bkNo или bkCustom.

TSpeedButton — кнопка панели инструментов

Кнопки TSpeedButton отличаются от кнопок TBitBtn тремя аспектами: во-первых, они могут фиксироваться в утопленном состоянии, во-вторых, они не могут закрыть модальное окно, в третьих, они не могут быть кнопками, выбранными по умолчанию. Во всем остальном они повторяют свойства и методы кнопок TBitBtn.

Для фиксации кнопка должна быть отнесена к какой-либо группе кнопок (эта группа может состоять из одной фиксируемой кнопки). Для этого используется свойство GroupIndex: Integer, которое не должно быть равно 0. Поведение кнопки определяется логическим свойством AllowAllUp: если это свойство имеет значение True, утопленная кнопка отпускается при фиксации любой другой

кнопки, входящей в ту же группу; если `AllowAllUp=False`, кнопку можно освободить только повторным щелчком. Индикатором состояния кнопки служит логическое свойство `Down`, которое имеет значение `True`, если кнопка утоплена. Свойство доступно для записи, что позволяет изменять состояние кнопки программно.

Для кнопки `TSpeedButton` предусмотрено событие `OnDbClick`, которое возникает при двойном щелчке на утопленной кнопке.

TMaskEdit — поле с маской ввода

Поле `TMaskEdit` предназначено для ввода текста, соответствующего некоторому шаблону, задаваемому свойством `EditMask: String`. Если это свойство не задано, компонент `TMaskEdit` работает как обычное текстовое поле `TEdit`. Свойство `IsMasked: Boolean` доступно только для чтения и содержит значение `True`, если строка шаблона задана. Свойство `EditText: String` содержит текст до наложения на него маски ввода (то есть то, что ввел пользователь), а свойство `Text: String` может (в зависимости от шаблона — см. ниже) содержать либо исходный текст, либо результат наложения на него маски.

Шаблон состоит из трех частей, отделенных друг от друга символами точки с запятой (;). Первая часть задает маску ввода, вторая — это символ 0 или 1, определяющий, нужно ли записывать в свойство `Text` результат наложения маски (1) или требуется оставить исходный текст (0). В третьей части указывается символ, который в поле с маской ввода будет присутствовать в местах, предназначенных для ввода символов (текст в поле может содержать символы маски). Например, для ввода семизначного номера телефона текст перед началом ввода может выглядеть так:

(095) XXX-XX-XX

Здесь доступные для ввода пользователя места обозначены символом X — последним символом в шаблоне.

Маска состоит из описателей мест ввода (табл. 16.2), специальных символов (табл. 16.3) и литералов. Описатель указывает, какой именно символ может ввести пользователь в данное место (описатель всегда указывает место для одного символа). Литерал вставляется в текст, показываемый в окне редактора, но при вводе курсор перескакивает через литерал и не дает пользователю возможности изменить его. Литералами считаются любые символы, кроме описателей полей и специальных символов, а также любой символ, которому предшествует символ \. Специальные символы формируют дополнительные указания редактору.

Таблица 16.2. Описатели мест ввода маски

Символ	Место ввода
L	Должно содержать букву
l	Может содержать букву
A	Должно содержать букву или цифру
a	Может содержать букву или цифру
C	Должно содержать любой символ

Символ	Место ввода
c	Может содержать любой символ
0	Должно содержать цифру
9	Может содержать цифру
#	Может содержать цифру, а также знаки + и -

Таблица 16.3. Специальные символы маски

Символ	Описание
\	Следующий символ — литерал. Позволяет вставлять в маску символы, относящиеся к описателям мест ввода и специальным символам
:	На это место вставляется символ-разделитель Windows для часов, минут, секунд
/	На это место вставляется символ-разделитель Windows для полей даты
;	Разделитель частей шаблона
!	Подавляет все начальные пробелы
>	Все следующие за этим символом места ввода преобразуют буквы в прописные
<	Все следующие за этим символом места ввода преобразуют буквы в строчные
<>	Отменяет преобразование букв

Примеры применения маски представлены в табл. 16.4.

Таблица 16.4. Примеры применения маски ввода

Шаблон с маской	Вид в редакторе	Свойство EditText	Свойство Text
(095) 000-0000;0;x	(095) xxx-xxxx	1234567	1234567
(095) 000-0000;1;x	(095) xxx-xxxx	1234567	(095)123-4567
(095)\0\00-0000;1;.	(095) 00.-....	12345	(095)001-2345

Если компонент лишается фокуса ввода, когда не все требуемые места ввода заполнены (или заполнены неправильно), возникает исключительная ситуация EDBEditError — на экран выводится окно с сообщением и после закрытия окна курсор устанавливается в то место ввода, на котором закончился правильный ввод.

Остальные свойства компонента TMaskEdit повторяют соответствующие свойства компонента TEdit. В частности, установка в свойство PasswordChar любого символа, кроме символа с кодом #0, обеспечивает секретность ввода: в этом случае все вводимые в компоненте символы заменяются символом, указанным в свойстве PasswordChar.

TStringGrid — текстовая таблица

Компонент TStringGrid предназначен для создания таблиц, в ячейках которых располагаются произвольные текстовые строки. Он является прямым потомком класса TDrawGrid (см. ниже), от которого им унаследовано большинство свойств и методов.

Таблица делится на две части — фиксированную и рабочую. Фиксированная часть служит для показа заголовков колонок и рядов, а также для ручного управления их размерами. Обычно фиксированная часть занимает левую колонку и верхний ряд таблицы, однако с помощью свойств `FixedCols` и `FixedRows` можно задать другое количество фиксированных колонок и рядов (если эти свойства имеют значение 0, таблица не содержит фиксированной зоны). Рабочая часть — это остальная часть таблицы. Она может содержать произвольное количество колонок и рядов, более того, эти величины могут изменяться программно. Рабочая часть может не уместиться целиком в пределах окна компонента, в этом случае в него автоматически помещаются нужные полосы прокрутки. При прокрутке рабочей области фиксированная область не исчезает, но меняется ее содержимое — заголовки колонок и рядов.

Центральным свойством компонента является свойство `Cells` — двумерный массив ячеек, каждая из которых может содержать произвольный текст. Конкретная ячейка определяется парой чисел — номером колонки и номером ряда, на пересечении которых она находится (нумерация начинается с нуля). Свойство `Cells` имеет тип **String**, поэтому программа может легко прочитать или записать содержимое нужной ячейки. Например:

```
Cells[1,1] := 'Левая верхняя ячейка рабочей зоны';
```

Количество ячеек по каждому измерению хранит пара свойств `ColCount` (количество колонок) и `RowCount` (количество рядов). Значения этих свойств и, следовательно, размеры таблицы могут меняться как на этапе разработки программы, так и в ходе ее работы, однако их значения должны быть как минимум на единицу больше соответственно значений в свойствах `FixedCols` и `FixedRows`, определяющих размеры фиксированной зоны.

Содержимое ячеек можно редактировать. Для этого в таблице используется специализированный потомок поля с маской ввода `TMaskEdit`.

Свойства компонента `TStringGrid` перечислены в табл. 16.5.

Таблица 16.5. Свойства компонента `TStringGrid`

Свойство	Описание
property <code>BorderStyle: TBorderStyle;</code>	Определяет рамку компонента: <code>bsNone</code> — нет рамки; <code>bsSingle</code> — рамка толщиной 1 пиксел
property <code>Cells[ACol, ARow: Integer]: String;</code>	Определяет содержимое ячейки с табличными координатами (<code>ACol, ARow</code>)
property <code>Col: LongInt;</code>	Содержит номер колонки с ячейкой, имеющей фокус ввода
property <code>ColCount: LongInt;</code>	Содержит количество колонок таблицы
property <code>Cols[Index: Integer]: TStrings;</code>	Содержит все строки колонки с индексом <code>Index</code>
property <code>ColWidths[Index: LongInt]: Integer;</code>	Содержит ширину колонки с индексом <code>Index</code>
property <code>DefaultColWidth: Integer;</code>	Содержит значение ширины колонки, заданное по умолчанию

Свойство	Описание
property DefaultDrawing: Boolean;	Разрешает/запрещает автоматическую прорисовку служебных элементов таблицы — фиксированной зоны, фона и прямоугольника ячейки, имеющей фокус ввода, и т. п.
property DefaultRowHeight: Integer;	Содержит значение высоты рядов, заданное по умолчанию
property EditorMode: Boolean;	Разрешает/запрещает редактирование ячеек. Игнорируется, если свойство Options включает значение goAlwaysShowEditor или не включает значение goEditing
property FixedColor: TColor;	Определяет цвет фиксированной зоны
property FixedCols: Integer;	Определяет количество колонок фиксированной зоны
property FixedRows: Integer;	Определяет количество рядов фиксированной зоны
property GridHeight: Integer;	Содержит значение высоты таблицы
property GridLineWidth: Integer;	Определяет толщину линий, расчерчивающих таблицу
property GridWidth: Integer;	Содержит значение ширины таблицы
property LeftCol: LongInt;	Содержит номер самого левого столбца, видимого в зоне прокрутки
property Objects [ACol, ARow: Integer]: TObject;	Обеспечивает доступ к объекту, связанному с ячейкой (ACol, ARow)
property Options: TGridOptions;	Содержит параметры таблицы (см. ниже)
property Row: LongInt;	Содержит номер ряда ячейки, имеющей фокус ввода
property RowCount: LongInt;	Содержит количество рядов таблицы
property RowHeights[Index: LongInt]: Integer;	Содержит значение высоты ряда с индексом Index
property Rows[Index: Integer]: TString;	Содержит все текстовые строки ряда с индексом Index
type TScrollStyle = (ssNone, ssHorizontal, ssVertical, ssBoth); property ScrollBars: TScrollStyle;	Определяет полосы прокрутки: ssNone — нет полос; ssHorizontal — в таблицу вставляется горизонтальная полоса; ssVertical — вставляется вертикальная полоса; ssBoth — вставляются обе полосы
TGridRect = record case Integer of 0: (Left, Top, Right, Bottom: LongInt); 1: (TopLeft, BottomRight: TGridCoord); end ; property Selection: TGridRect;	Определяет группу выделенных ячеек в координатах левой верхней и правой нижней ячейки (нумерация колонок и рядов идет от нуля, включая колонки и ряды фиксированной зоны). После выделения фокус ввода окажется в правой нижней ячейке

Таблица 16.5 (продолжение)

Свойство	Описание
property TabStops[Index: LongInt]: Boolean;	Разрешает/запрещает выбирать колонку с индексом Index при обходе ячеек с помощью клавиши Tab. Игнорируется, если свойство Options не содержит значения goTabs
property TopRow: LongInt;	Содержит номер самого верхнего ряда, видимого в прокручиваемой зоне ячеек
property VisibleColCount: Integer;	Содержит количество колонок, полностью видимых в зоне прокрутки
property VisibleRowCount: Integer;	Содержит количество рядов, полностью видимых в зоне прокрутки

Для компонента определен тип TGridOptions:

type

```
TGridOption = (goFixedVertLine, goFixedHorzLine,
  goVertLine, goHorzLine, goRangeSelect,
  goDrawFocusSelected, goRowSizing, goColSizing,
  goRowMoving, goColMoving, goEditing, goTabs,
  goRowSelect, goAlwaysShowEditor, goThumbTracking);
TGridOptions = set of TGridOptions;
```

Назначение элементов множества TGridOptions представлено в табл. 16.6.

Таблица 16.6. Назначение элементов множества TGridOptions

Элемент	Описание
goFixedVertLine	Колонки фиксированной зоны разделяются вертикальными линиями
goFixedHorzLine	Ряды фиксированной зоны разделяются горизонтальными линиями
goVertLine	Колонки рабочей зоны разделяются вертикальными линиями
goHorzLine	Колонки рабочей зоны разделяются горизонтальными линиями
goRangeSelect	Разрешено выделение нескольких ячеек. Игнорируется, если во множество включен элемент goEditing
goDrawFocusSelected	Разрешено показывать ячейку, имеющую фокус ввода, так же, как выделенную ячейку
goRowSizing	Разрешено ручное (мышью) изменение высоты рядов
goColSizing	Разрешено ручное изменение ширины колонок
goRowMoving	Разрешено ручное перемещение рядов (нажать левую кнопку мыши на фиксированной ячейке перемещаемого ряда и, удерживая кнопку нажатой, перетащить ряд на новое место)

Элемент	Описание
goColMoving	Разрешено ручное перемещение колонок
goEditing	Разрешено редактирование ячейки. Игнорируется, если включен элемент goRowSelect. Редактирование начинается после щелчка мышью или нажатия клавиши F2 и завершается при щелчке на другой ячейке или нажатии клавиши Enter
goTabs	Разрешено обходить ячейки клавишей Tab (Shift+Tab)
goRowSelect	Обязывает выделять сразу все ячейки ряда и запрещает редактирование ячеек
goAlwaysShowEditor	Разрешено редактировать ячейку, имеющую фокус ввода: редактирование возможно после выделения ячейки клавишей Tab (Shift+Tab). Игнорируется, если не включен элемент goEditing
goThumbTracking	Разрешено обновление при прокрутке. Если этот элемент отсутствует, обновление ячеек произойдет только после окончания прокрутки

Два метода класса могут оказаться полезными для процедуры прорисовки (табл. 16.7).

Таблица 16.7. Методы компонента TStringGrid

Метод	Описание
function CellRect (ACol, ARow: LongInt): TRect;	Возвращает прямоугольник ячейки по номерам колонки ACol и ряда ARow
procedure MouseToCell (X, Y: Integer; var ACol, ARow: LongInt);	Возвращает табличные координаты ячейки ACol и ARow по экранным координатам (X,Y) точки

Остальные методы рассчитаны на перекрытие в потомках и представляют интерес только для разработчиков новых компонентов. Для программистов-пользователей важны доступные компоненту события, перечисленные в табл. 16.8.

Таблица 16.8. События компонента TStringGrid

Событие	Описание
TMovedEvent = procedure (Sender: TObject; FromIndex, ToIndex: LongInt) of object ; property OnColumnMoved: TMovedEvent;	Возникает при перемещении колонки с индексом FromIndex в положение, определяемое индексом ToIndex
TGridDrawState = set of (gdSelected, gdFocused, gdFixed); TDrawCellEvent = procedure (Sender: TObject; Col, Row: LongInt; Rect: TRect; State:	Возникает при необходимости перерисовать ячейку с табличными координатами (Col,Row): Rect — прямоугольник прорисовки; State — состояние ячейки (gdSelected — ячейка выделена; gdFocused — ячейка имеет фокус ввода; gdFixed — ячейка принадлежит

продолжение ➤

Таблица 16.8. (продолжение)

Событие	Описание
<code>TGridDrawState) of object;</code> property <code>OnDrawCell:</code> <code>TDrawCellEvent;</code>	фиксированной зоне таблицы). Для прорисовки используется табличное свойство <code>Canvas</code>
<code>TGetEditEvent = procedure</code> <code>(Sender: TObject; ACol,</code> <code>ARow: LongInt; var Value:</code> <code>String) of object;</code> property <code>OnGetEditMask:</code> <code>TGetEditEvent;</code>	Возникает при редактировании текста в ячейке с табличными координатами (<code>ACol,ARow</code>). В параметре <code>Value</code> обработчик должен вернуть шаблон для компонента <code>TEditMask</code>
property <code>OnGetEditText:</code> <code>TGetEditEvent;</code>	Возникает при редактировании текста в ячейке с табличными координатами (<code>ACol,ARow</code>). В параметре <code>Value</code> обработчик должен вернуть текст для компонента <code>TEditMask</code> (см. событие <code>OnGetEditMask</code>)
property <code>OnRowMoved:</code> <code>TMovedEvent;</code>	Возникает при перемещении ряда с индексом <code>FromIndex</code> в положение, определяемое индексом <code>ToIndex</code> (см. событие <code>OnColMoved</code>)
<code>SelectCellEvent = procedure</code> <code>(Sender: TObject; Col, Row:</code> <code>LongInt; var CanSelect:</code> <code>Boolean) of object;</code> property <code>OnSelectCell:</code> <code>TSelectCellEvent;</code>	Возникает при попытке выделить ячейку с табличными координатами (<code>Col,Row</code>). В параметре <code>CanSelect</code> обработчик сообщает о возможности выделения ячейки
<code>TSetEditEvent = procedure</code> <code>(Sender: TObject; ACol,</code> <code>ARow: LongInt; const Value:</code> <code>String) of object;</code> property <code>OnSetEditText:</code> <code>TSetEditEvent;</code>	Возникает при завершении редактирования ячейки (<code>ACol,ARow</code>). В параметре <code>Value</code> обработчик получает результат ввода или редактирования текста
property <code>OnTopLeftChanged:</code> <code>TNotifyEvent;</code>	Возникает после изменения значения <code>TopRow</code> или <code>LeftCol</code> в результате прокрутки рабочей зоны

В следующем примере (проект `Chap_16\StringGrid\StringGrid.dpr`) компонент `TStringGrid` используется для отображения всех слов из произвольного текстового файла.

1. Назовите пустую форму именем `fmStGrid` и поместите на нее панель `TPanel`.
2. Установите в свойство `Align` панели значение `alBottom`, чтобы панель всегда занимала самую нижнюю часть окна, и удалите текст в ее свойстве `Caption`.
3. Поместите на панель кнопку `TBitBtn`, установите в ее свойство `Caption` строку Открыть файл... загрузите в свойство `Glyph` файл `IMAGES\BUTTONS\FILEOPEN.BMP` и растяните кнопку по горизонтали так, чтобы надпись и изображение полностью умещались на ее поверхности.
4. Поместите на панель еще одну кнопку `TBitBtn` и установите в ее свойство `Kind` значение `bkClose`, а в свойство `Caption` — строку Выход.

5. Поместите на панель компонент `OpenDialog` с вкладки `Dialogs` палитры компонентов. Этот компонент создает и обсуживает стандартное для Windows окно открытия файла. С его помощью при прогоне программы вы сможете выбрать текстовый файл (например, файл с текстом программы), чтобы программа показала в таблице все слова из этого файла.
6. На форму (а не на панель `Panel1`) поместите компонент `TStringGrid`.
7. Поместите в свойства `ColCount` и `RowCount` компонента `TStringGrid` значения 2 (количество колонок и рядов таблицы должно быть больше, чем количество фиксированных колонок `FixedCols` и количество фиксированных рядов `FixedRows`). Установите в свойство `Align` компонента значение `alClient`, чтобы он занял все свободное место формы.

На этом закончим этап разработки формы. Вся основная работа будет связана с обработкой щелчка на кнопке `BitBtn1`: программа сначала предложит вам выбрать текстовый файл, затем прочитает его и создаст список отсортированных в алфавитном порядке слов из каждой строки файла, после чего наполнит словами таблицу. Наполнение будет происходить так, чтобы каждая колонка строки содержала все слова, начинающиеся на одну и ту же букву (рис. 16.2).

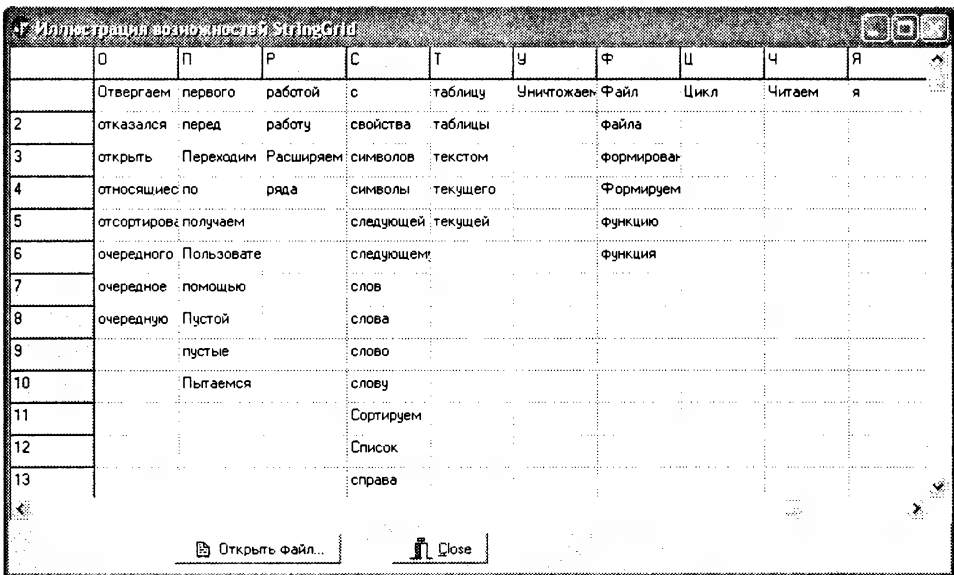


Рис. 16.2. Пример использования компонента `TStringList`

Обработчик события `OnClick` для кнопки `BitBtn1` показан в листинге 16.1.

Листинг 16.1. Обработчик события `OnClick`¹

```
procedure TfmStGrid.BitBtn1Click(Sender: TObject);
Function GetWord(var S: String): String;
```

¹ Все представленные в книге тексты программ можно загрузить с сайта издательства по адресу <http://www.piter.com/download> — Примеч. ред.


```

{ Вспомогательная функция для выделения очередного слова
из строки }
const    // Множество символов слова:
    Letters: set of Char = ['a'..'z', 'A'..'Z', 'А'..'Я'];
begin
    Result := '';
    // Удаляем в начале строки все символы, не относящиеся
    // к слову
    while (S <> '') and not (S[1] in Letters) do
        Delete(S, 1, 1);
    // Формируем очередное слово
    while (S <> '') and (S[1] in Letters) do
        begin
            Result := Result + S[1];
            Delete(S, 1, 1)
        end;
end; // GetWord

var
    F: TextFile;           // Файл с текстом
    S, Word: String;      // Вспомогательные строки
    NCol, NRow: Integer;   // Номер текущей колонки и текущего
                           // ряда
    Words: TStringList;    // Список отсортированных слов
                           // из файла

begin
    // С помощью стандартного диалогового окна получаем
    // имя файла
    if not OpenDialog1.Execute then
        Exit; // Пользователь отказался выбрать файл
    // Пытаемся открыть файл
    AssignFile(F, OpenDialog1.FileName);
    try
        Reset(F);
    except
        // Файл нельзя открыть:
        ShowMessage('Невозможно открыть файл ' +
            OpenDialog1.FileName);
        Exit;
    end;
    // Готовим список Words:
    Words := TStringList.Create;
    Words.Sorted := True; // Сортируем строки
    Words.Duplicates := dupIgnore; // Отвергаем дубликаты
    // Изменяем указатель мыши перед длительной работой
    Screen.Cursor := crHourGlass;

```

```

// Читаем файл по строкам
while not EOF(F) do
begin
  ReadLn(F, S);           // Читаем очередную строку
  // Выделяем из строки слова и заносим их в список Words
  while S <> '' do
  begin
    Word := GetWord(S);
    if Word <> '' then
      Words.Add(Word)      // Не вставляем пустые строки
  end
end; // while not EOF(F) do
Screen.Cursor := crDefault; // Восстанавливаем указатель
CloseFile(F);           // Закрываем файл
if Words.Count=0 then
  Exit;                  // Пустой файл — выходим
with TStringGrid1 do
begin
  NCol := 1;              // Номер первой колонки слов
  // Цикл формирования таблицы
  while Words.Count > 0 do
  begin
    // Формируем заголовок колонки
    // и начальное значение номера ряда
    Cells[NCol, 0] := AnsiUpperCase(Words[0][1]);
    NRow := 0;
    // Цикл заполнения очередной колонки
    while (Words.Count > 0) and
      (AnsiUpperCase(Words[0][1]) = Cells[NCol, 0]) do
    begin
      inc(NRow);          // Номер текущего ряда
      if NRow = RowCount then
      begin              // Расширяем таблицу вниз
        RowCount := RowCount + 1; { Для свойства RowCount
                                   нельзя использовать
                                   функцию инкремента inc! }
        Cells[0, NRow] := IntToStr(NRow);
      end;
      Cells[NCol, NRow] := Words[0];
      Words.Delete(0);
    end; // while (Words.Count > 0)
    // Переходим к следующей колонке
    if Words.Count=0 then
      Break;             // Кончаем работу, если слов больше нет
    inc(NCol);           // Переходим к следующей колонке
    ColCount := ColCount+1 // Расширяем таблицу вправо
                        // на 1 колонку
  end;
end;

```

```

    end; // while Words.Count > 0 do
  end; // with StringGrid1 do
end; // procedure TfmStGrid.BitBtn1Click

```

TDrawGrid — произвольная таблица

Компонент TDrawGrid предоставляет программисту мощные возможности создания и обслуживания табличных структур данных. Он обеспечивает двухмерное представление данных, упорядоченных по колонкам и рядам, и избавляет программиста от многих рутинных операций, связанных с представлением и обслуживанием таблиц.

Чтобы таблица была работоспособной, в ней как минимум следует определить обработчик события OnDrawCell, которое возникает при необходимости прорисовать ту или иную ячейку. Для прорисовки используется табличное свойство Canvas.

Компонент TDrawGrid является непосредственным родителем строковой таблицы TStringGrid, поэтому передает ей все свои свойства, методы и события за исключением специфичных для строк свойств Cells, Cols, Objects и Rows. Поскольку порядок описания компонентов в этой главе соответствует порядку их расположения на вкладках палитры компонентов, компонент TStringGrid описан раньше своего родителя TDrawGrid, и вы сможете найти информацию о свойствах, методах и событиях последнего в предыдущем разделе.

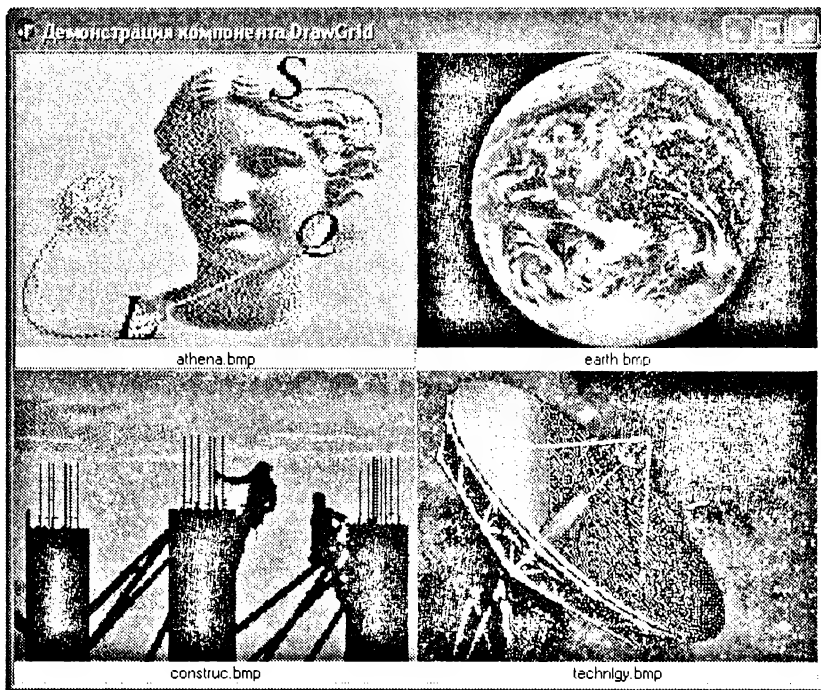


Рис. 16.3. Пример использования таблицы TDrawGrid для отображения картинок и текста

В следующем примере (проект Chap_16\DrawGrid\DrawGrid.dpr) компонент TDrawGrid используется для одновременного отображения текста и картинок, причем текст можно редактировать. Картинки взяты из каталога Images\Splash\16color (рис. 16.3).

Для получения окна, показанного на рисунке, расположите на пустой форме fmDrawGrid компонент TDrawGrid и задайте для него значения свойств, перечисленные в табл. 16.9.

Таблица 16.9. Значения свойств компонента TDrawGrid

Свойство	Значение	Комментарий
Align	alClient	Таблица занимает всю клиентскую часть окна
ColCount	2	В таблице 2 колонки
FixedCols	0	Нет фиксированных колонок
FixedRows	0	Нет фиксированных рядов
Name	dgDraw	Имя таблицы
Options.goEditing	True	Разрешено редактирование текста
RowCount	2	В таблице 2 строки
ScrollBars	sbNone	Не надо вставлять полосы прокрутки

Исходный код файла DrawGridMain.pas представлен в листинге 16.2.

Листинг 16.2. Иллюстрация создания компонента TDrawGrid

```

unit DrawGridMain;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls,
    Forms, Dialogs, Grids;

type
    TfmDrawGrid = class(TForm)
        dgDraw: TDrawGrid;
        procedure FormCreate(Sender: TObject);
        procedure FormDeactivate(Sender: TObject);
        procedure dgDrawGetEditText(Sender: TObject;
            ACol, ARow: Integer;
            var Value: String);
        procedure dgDrawSetEditText(Sender: TObject;
            ACol, ARow: Integer;
            const Value: String);
        procedure dgDrawDrawCell(Sender: TObject;
            ACol, ARow: Integer;
            Rect: TRect;

```

```

                                State: TGridDrawState);
  procedure FormResize(Sender: TObject);
  private
    { Private declarations }
    slBitMap: TStringList; // Список подписей и изображений
  public
    { Public declarations }
end;

var
  fmDrawGrid: TfmDrawGrid;

implementation

{$R *.DFM}
procedure TfmDrawGrid.FormCreate(Sender: TObject);
// Создает и наполняет список slBitMap в момент создания
// формы
const
  Bmp_Path = 'C:\Program Files\Common Files\Borland
    Shared\'+'Images\Splash\16Color\';
  FNames: array [0..3] of String =           // Имена файлов
                                              // с картинками
    ('athena.bmp', 'construc.bmp', 'earth.bmp',
     'technlgy.bmp');
var
  k: Integer;
begin
  // Создаем список строк:
  slBitMap := TStringList.Create;
  // Вставляем в него названия файлов и картинки:
  with slBitMap do for k := 0 to 3 do
  begin
    Add(FNames[k]);
    Objects[k] := TBitmap.Create;
    (Objects[k] as TBitmap).LoadFromFile(Bmp_Path+
      FNames[k])
  end // with slBitMap do for k := 0 to 3 do
end; // FormCreate

procedure TfmDrawGrid.FormDeactivate(Sender: TObject);
// Удаляем список slBitMap при разрушении формы
begin
  slBitMap.Free
end;

```

```

procedure TfmDrawGrid.dgDrawGetEditText(Sender: TObject;
                                         ACol, ARow: Integer;
                                         var Value: String);
// Передает в редактируемое поле текст из ячейки
begin
    Value := slBitMap[2 * ACol + ARow]
end;

procedure TfmDrawGrid.dgDrawSetEditText(Sender: TObject;
                                         ACol, ARow: Integer; const Value:
                                         String);
// Получает текст из редактируемого поля
// и сохраняет его в slBitMap
begin
    slBitMap[2 * ACol + ARow] := Value
end;

procedure TfmDrawGrid.dgDrawDrawCell(Sender: TObject;
                                       ACol, ARow: Integer;
                                       Rect: TRect;
                                       State:
TGridDrawState);
// Рисует картинку из slBitMap.Objects и текст из slBitMap
var
    k: Integer;
begin
    // Пересчитываем координаты ячейки в индекс списка:
    k := 2 * ACol + ARow;
    with dgDraw.Canvas, Rect, slBitMap do
        begin
            // Уменьшаем высоту картинки для вывода под ней текста:
            Bottom := Bottom - TextHeight('1') - 2;
            // Рисуем картинку:
            StretchDraw(Rect, (Objects[k] as TBitmap));
            // Центрируем текст по горизонтали и выводим его:
            TextOut(Left + (Right - Left - TextWidth(slBitMap[k])) div 2,
                    Bottom + 1, slBitMap[k])
        end // with dgDraw.Canvas, Rect, slBitMap do
end; // dgDrawDrawCell

procedure TfmDrawGrid.FormResize(Sender: TObject);
// Изменяет размеры ячеек при изменении размеров окна.
// Запас в один пиксел необходим для предотвращения
// автоматической прокрутки
begin

```

```

with dgDraw do
begin
  DefaultColWidth := ClientRect.Right div 2 - 1;
  DefaultRowHeight := ClientRect.Bottom div 2 - 1
end;
end;
end.

```

Для оперативного хранения изображений и подписей к ним используется следующий список строк:

```

private
  slBitmap: TSringList;

```

Этот список необходимо создать и наполнить в момент создания окна и удалить при его уничтожении. Эти операции осуществляются в обработчиках событий `OnCreate` и `OnDestroy` формы. Обратите внимание: в константе `Bmp_Path` указывается стандартный путь доступа к графическим файлам. Если на вашем компьютере они установлены в другой папке, эту константу нужно соответствующим образом изменить, после этого полезно осуществить прогон программы и убедиться в доступности файлов.

Чтобы отобразить текст подписи в окне редактора (оно появляется автоматически после щелчка на подписи), необходимо создать обработчик события `OnGetEditText`. В нем строка списка `slBitmap`, индекс которой вычисляется на основе текущих колонки и ряда табличной ячейки, передается встроенному в компонент редактору. Поскольку в программе разрешается редактировать текст подписи, обработчик `OnSetEditText` осуществляет обратные действия и сохраняет в списке `slBitmap` отредактированную подпись. Обработчик `OnDrawCell` обязательно создается для любого компонента `TDrawGrid`, так как без него компонент не сможет заполнить пустые ячейки каким-либо изображением и/или текстом.

Чтобы согласовать размеры ячеек таблицы с размерами окна программы, используется обработчик события `OnResize` (это событие возникает при каждом изменении размеров окна). Высота (ширина) каждой ячейки устанавливается равной половине высоты (ширины) клиентской области формы минус 1 пиксел. Этот запас предотвращает автоматическую вставку в таблицу полос прокрутки.

TImage — изображение

Компонент `TImage` служит для размещения на форме одного из трех поддерживаемых Delphi типов изображений: растрового рисунка, значка или метафайла. Любой из этих типов изображений содержится в центральном свойстве компонента — `Picture`. Свойство `Canvas` открывает доступ к канве, с помощью которой при необходимости программа может отредактировать растровое изображение (но не значок или метафайл!).

Вот как можно создать несложную программу (проект `Chap_16\TImage\Image-Demo.dpr`) просмотра изображений, окно которой показано на рис. 16.4.

1. Поместите на форму панель `TPanel`, очистите ее свойство `Caption` и введите значение `alBottom` в свойство `Alignment` — панель займет нижнюю часть формы.

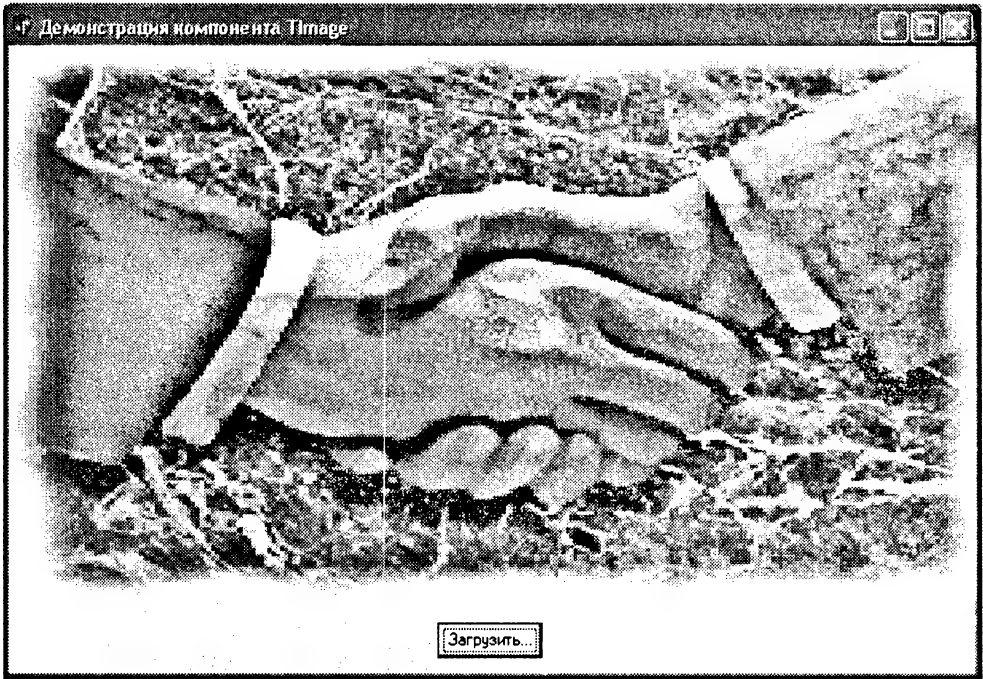


Рис. 16.4. Компонент TImage воспроизводит изображение

2. Разместите на панели кнопку TButton с надписью Загрузить.
3. На любое свободное место формы (над панелью) поместите компонент TImage и установите в его свойство Align значение alClient.
4. На любое место (TImage, TPanel или даже TButton) поместите компонент TOpenPictureDialog (вкладка Dialogs палитры компонентов) — с его помощью пользователь вашей программы сможет разыскать нужный файл с изображением.
5. Для кнопки Button1 напишите следующий обработчик события OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then
    begin
      Image1.Picture.LoadFromFile
        (OpenPictureDialog1.FileName);
      Image1.Stretch := True
    end
end;
```

Замечу, что показанный ниже оператор масштабирует изображение так, чтобы оно занимало всю рабочую область TImage:

```
Image1.Stretch := True
```

Свойства компонента TImage перечислены в табл. 16.10.

Таблица 16.10. Свойства компонента TImage

Свойство	Описание
property AutoSize: Boolean;	Разрешает/запрещает изменять размеры компонента так, чтобы в них полностью загрузилось изображение (см. свойство Stretch)
property Canvas: TCanvas;	Содержит канву для прорисовки изображения
property Center: Boolean;	Указывает, надо ли центрировать изображение в границах компонента. Игнорируется, если AutoSize=True или если Stretch=True и изображение не является значком (ICO)
property Picture: TPicture;	Центральное свойство класса. Служит контейнером изображения
property Stretch: Boolean;	Разрешает/запрещает изменять размер изображения так, чтобы оно целиком заполнило клиентскую область компонента (см. свойство AutoSize)
property Transparent: Boolean;	Запрещает/разрешает накладывать собственный фон изображения на фон компонента

С помощью методов класса TPicture (см. подраздел «Классы TGraphic и TPicture» раздела «Графический инструментарий» в главе 13), к которому относится свойство Picture, программист может загрузить изображение из файла или буфера обмена (clipboard) и сохранить его в файле или буфере.

TShape — стандартная фигура

Компонент рисует одну из простейших геометрических фигур, определяемых следующим перечисленным типом (прямоугольник, квадрат, скругленный прямоугольник, скругленный квадрат, эллипс, окружность):

```
type TShapeType = (stRectangle, stSquare, stRoundRect,
                   stRoundSquare, stEllipse, stCircle);
```

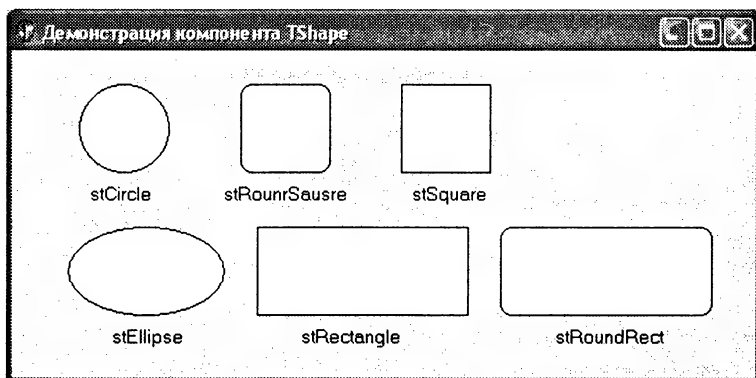


Рис. 16.5. Стандартные фигуры

Фигура (рис. 16.5) полностью занимает все пространство компонента. Если задан квадрат или круг, а размеры элемента по горизонтали и вертикали отличаются, фигура чертится с размером меньшего измерения.

Помимо стандартных графических инструментов `Brush` и `Pen` (шрифт для компонента не нужен), в компоненте определено свойство `Shape`: `TShapeType`, которое и задает вид геометрической фигуры. Изменение этого свойства приводит к немедленной перерисовке изображения.

TBevel — кромка

Компонент класса `TBevel` носит оформительский характер и предназначен для визуального выделения группы элементов или отделения их друг от друга. Вид компонента (прямоугольник, рамка, верхняя линия, нижняя линия, левая линия, правая линия) определяет следующее свойство:

```
type TBevelShape = (bsBox, bsFrame, bsTopLine,
                    bsBottomLine, bsLeftLine, bsRightLine);
```

```
property Shape: TBevelShape;
```

Стиль компонента (вдавленный или выпуклый) определяет другое свойство:

```
type TBevelStyle = (bsLowered, bsRaised);
```

```
property Style: TBevelStyle;
```

TScrollBar — панель с полосами прокрутки

Компонент класса `TScrollBar` служит контейнером для размещения других компонентов. Его отличительная особенность — возможность прокрутки и, следовательно, экономия пространства формы при необходимости размещения на ней большого количества элементов управления. Замечу, что в современных программных продуктах относительно редко используются такого рода компоненты, предпочтение отдается многостраничным элементам управления с вкладками. Наиболее подходящей областью применения компонентов является размещение на них относительно больших по размерам компонентов `TEdit`, `TComboBox`, `TMemo` и т. п.

Использование компонента не отличается сложностью: поместите его на форму и размещайте затем на нем другие компоненты. Если очередной компонент выйдет за пределы рабочей области компонента `TScrollBar`, по сторонам контейнера возникнут полосы прокрутки (рис. 16.6).

Однако если свойство `AutoScroll` компонента содержит значение `False`, полосы прокрутки не появятся, и компонент будет отсекаать «лишние» части своих дочерних компонентов. Еще одним важным свойством компонента является `AutoSize`: если это свойство имеет значение `True`, размеры компонента будут автоматически изменяться так, чтобы все части дочерних компонентов оставались в его рабочей зоне, даже если для этого придется увеличить размеры внешнего контейнера (формы). С помощью свойств `HorzScrollBar` и `VertScrollBar` программист может управлять свойствами каждой из полос прокрутки в отдельности.

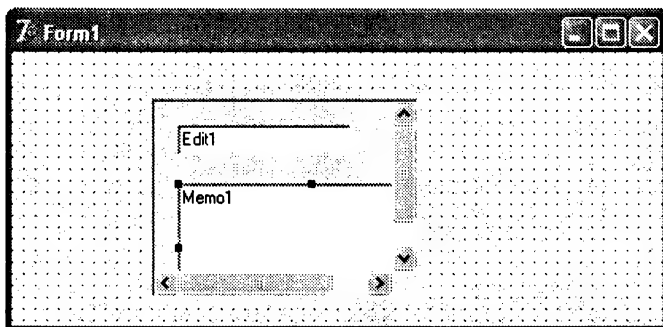


Рис. 16.6. Размещение компонентов на компоненте TScrollBox

TCheckBox — группа флажков

Подобно тому, как компонент TRadioGroup группирует переключатели, компонент TCheckBox группирует флажки, позволяя обратиться к любому из них по индексу.

Чтобы вставить в компонент группу флажков, следует раскрыть редактор его свойства Items и ввести одну или несколько текстовых строк — каждая строка будет соответствовать поясняющей надписи справа от флажка.

Свойства компонента TCheckBox перечислены в табл. 16.11.

Таблица 16.11. Свойства компонента TCheckBox

Свойство	Описание
property AllowGrayed: Boolean;	Разрешает/запрещает использовать третье состояние флажка cbGrayed (не уверен)
property BorderStyle: TBorderStyle;	Определяет тип рамки, очерчивающей компонент: bsNone — нет рамки; bsSingle — рамка толщиной в 1 пиксел
property Canvas: TCanvas;	Это свойство используется для программной прорисовки флажков
property Checked[Index: Integer]: Boolean;	Содержит для флажка с индексом Index значение, идентифицирующее выбор пользователя. Для состояний cbUnchecked и cbGrayed свойство имеет значение False
property Columns: Integer;	Определяет количество колонок флажков
property IntegralHeight: Boolean;	Если имеет значение True, компонент автоматически изменяет высоту так, чтобы в нем полностью отображались все флажки
property ItemHeight: Integer;	Если свойство Style компонента имеет значение lsOwnerDrawFixed, значение ItemHeight определяет высоту каждого флажка компонента. При других значениях Style это свойство игнорируется

Свойство	Описание
property ItemIndex: Integer;	Содержит индекс флажка, состояние которого изменилось (индексация флажков начинается с нуля)
property Items: TStrings;	Содержит подписи флажков компонента
property MultiSelect: Boolean;	Если содержит значение True, пользователь может устанавливать несколько флажков
property SelCount: Integer;	Содержит количество установленных флажков, если свойство MultiSelect содержит значение True
property Selected[Index: Integer]: Boolean;	Содержит значение True, если установлен флажок с индексом Index
property Sorted: Boolean;	Сортирует по алфавиту подписи к флажкам
property State[Index: Integer]: TCheckBoxState;	Содержит состояние флажка с индексом Index: cbUnchecked — не установлен; cbChecked — установлен; cbGrayed — не уверен
type TListBoxStyle = (lbStandard, lbOwnerDrawFixed, lbOwnerDrawVariable); property Style: TListBoxStyle;	Определяет способ прорисовки флажков: lbStandard — флажки прорисовывает Windows; lbOwnerDrawFixed — компонент использует программный метод прорисовки флажков, каждый из которых имеет одинаковую высоту ItemHeight; lbOwnerDrawVariable — флажки прорисовываются программно и могут иметь разную высоту
property TabWidth: Integer;	Определяет ширину табуляционного пробела
property TopIndex: Integer;	Содержит индекс самого первого отображаемого флажка

Для компонента определено событие, которое наступает при изменении состояния любого флажка:

property OnClickCheck: TNotifyEvent;

TSplitter — вешка разбивки

Компонент TSplitter предназначен для ручного (с помощью мыши) управления размерами контейнеров TPanel, TGroupBox и им подобных во время прогона программы. Визуально он представляет собой небольшую вертикальную или горизонтальную полосу, располагающуюся между двумя соседними контейнерами или на «свободной» стороне одного из них.

Непременным условием правильной работы компонента является выравнивание контейнера (контейнеров), размерами которого (которых) он управляет, вдоль одной из сторон окна или охватывающего контейнера. Для примера на рис. 16.7 показаны две панели, разделенные компонентом TSplitter. Для нижней панели выполняется условие Align=alBottom, верхняя панель в этом случае может иметь

выравнивание `alBottom` или `alClient`. Между панелями помещен компонент `TSplitter` с выравниванием `alBottom`.

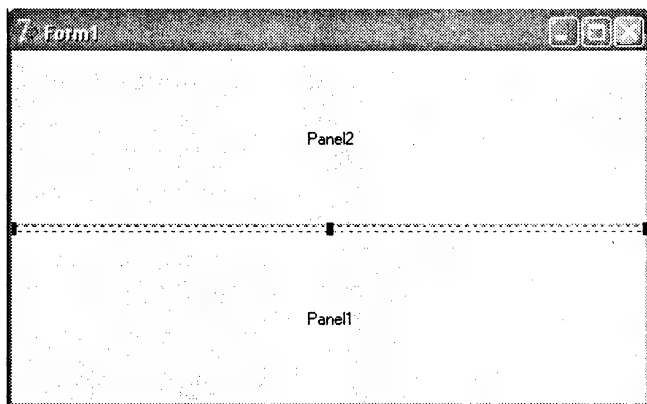


Рис. 16.7. Пример использования компонента `TSplitter`

Свойства компонента `TSplitter` представлены в табл. 16.12.

Таблица 16.12. Свойства компонента `TSplitter`

Свойство	Описание
property Beveled: Boolean;	Управляет трехмерным изображением компонента. Если имеет значение <code>False</code> , компонент виден как узкая полоска фона между разделяемыми им компонентами
<code>NaturalNumber = 1..High(Integer);</code> property MinSize: NaturalNumber;	Содержит минимальный размер любого из компонентов, которых разделяет компонент <code>TSplitter</code> . Если выравнивание имеет значение <code>alLeft</code> или <code>alRight</code> — минимальная ширина компонента слева и справа от вешки разбивки, если <code>alTop</code> или <code>alBottom</code> — минимальная высота компонента выше или ниже от нее

Замечу, что в отличие от других видимых компонентов, `TSplitter` не имеет свойства `ShowHint`, поэтому он не может отображать оперативную подсказку в небольшом окне. Однако у него есть свойство `Hint`, содержимое которого (до возможного разделителя в виде вертикальной черты |) может отображать обработчик события `Application.OnHint` (см. раздел «Поддержка справочной службы» в главе 14).

Для компонента определено событие `OnMoved`, которое вызывается при любом перемещении компонента мышью.

TStaticText — текстовая метка

Компонент `TStaticText` во всем подобен своему «двоюродному брату» `TLabel` за исключением двух аспектов. Во-первых, он порожден от класса `TWinControl` и, таким образом, имеет Windows-окно. Это обстоятельство может быть необходимым условием правильного взаимодействия с внешним для Delphi элементом

ActiveX, с которым текстовая метка связана своим свойством `FocusControl`. Во-вторых, в его свойстве `BorderStyle` добавлено значение `sbsSunken`, которое создает иллюзию рельефности компонента:

```
type TStaticBorderStyle = (sbsNone, sbsSingle, sbsSunken);
property BorderStyle: TStaticBorderStyle;
```

Все остальные свойства и методы компонента совпадают со свойствами и методами `TLabel`. Отмечу, что компонент обладает внешней и внутренней рамками, так что его вид можно разнообразить.

TControlBar — контейнер для панели инструментов

Компонент `TControlBar` служит удобным контейнером для размещения панелей инструментов `TToolBar` (см. раздел «`TToolBar` и `TToolButton` — панель инструментов и кнопки для нее» в главе 17). Он активно использует технологию «причаливания» (`Drag&Dock`) для управления положением панелей. На рис. 16.8 показан компонент `TControlBar` с двумя помещенными на него панелями `TToolBar`.

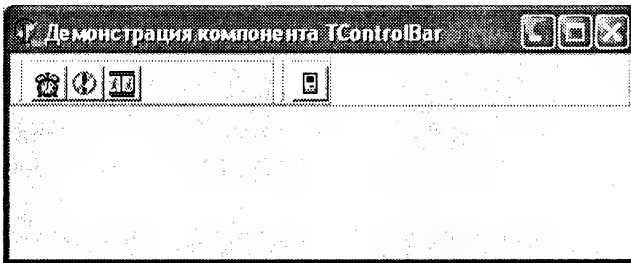


Рис. 16.8. Пример использования компонента `TControlBar`

Если вы захотите повторить пример, проделайте следующее.

1. Создайте новое приложение с пустой формой и поместите на нее компонент `TControlBar`.
2. Установите в его свойство `Align` значение `alTop`, в `AutoSize` — `True`, в `DragKind` — `dkDock`. Оставьте в свойстве `DragMode` значение `dmManual`.
3. Поместите на свободное место формы два компонента `TImageList` (вкладка `Win32` палитры компонентов) и наполните каждый из них двумя-тремя изображениями. Чтобы наполнить компонент `TImageList` изображениями, откройте редактор изображений, дважды щелкнув на компоненте мышью или выбрав в его контекстном меню (щелчок правой кнопкой) команду `ImageList Editor`.
4. В появившемся окне редактора щелкните на кнопке `Add` и выберите любой BMP-файл в каталоге `Images\Buttons`. Файлы в этом каталоге имеют по два изображения, в то время как компонент `TImageList` способен хранить только одиночные изображения одинакового размера. Поэтому редактор предложит вам разделить два изображения — согласитесь с ним (кнопка `OK`). Щелкните на правом изоб-

ражении в окне редактора (оно предназначено для представления кнопки в недоступном состоянии) и щелкните на кнопке Delete, чтобы удалить его.

- Повторите предыдущий шаг для загрузки нескольких изображений (рис. 16.9) и затем закройте окно редактора.

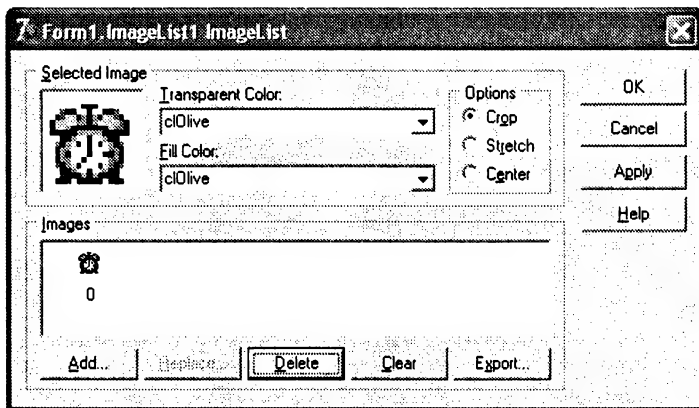


Рис. 16.9. Окно редактора ImageList

- Поместите на компонент ControlBar1 компонент TToolBar (вкладка Win32).
- Установите в свойство AutoSize компонента ToolBar1 значение True, в свойство Images — значение ImageList1 и поместите в него столько кнопок, сколько изображений вы подготовили в ImageList1. Для вставки кнопки щелкните на компоненте ToolBar1 правой кнопкой мыши и выберите в контекстном меню команду New Button.
- Поместите на свободное место формы еще один компонент TToolBar. Он автоматически прижмется к нижнему краю компонента ControlBar1. Измените в его свойстве Align значение alTop на alNone и мышью перетащите компонент несколько вниз — на этапе прогона программы этот компонент будет «причаливать» к компоненту ControlBar1. Свяжите компонент TToolBar с компонентом ImageList2 (свойство Images) и создайте в нем кнопки (см. предыдущий шаг).
- Для обоих компонентов поместите в свойство DragKind значение dkDock, в DragMode — dmAutomatic и в AutoSize — True.
- Запустите программу и перетащите нижнюю панель инструментов на компонент ControlBar1.

Свойства компонента TControlBar представлены в табл. 16.13.

Таблица 16.13. Свойства компонента TControlBar

Свойство	Описание
property AutoDrag: Boolean;	Разрешает/запрещает компонентам TToolBar покидать границы компонента TControlBar

Свойство	Описание
property Picture: TPicture;	Содержит изображение, которое, периодически повторяясь, создает фон компонента
property RowSize: TRowSize;	Задает высоту одного ряда панелей инструментов. Значение по умолчанию 25 в точности соответствует высоте одной панели, и в этом случае между рядами нет зазора
property RowSnap: Boolean;	Если содержит True, «причаливаемая» панель будет выравниваться по высоте ряда

Следующий метод устанавливает все дочерние компоненты на их окончательные позиции после «причаливания» или «отчаливания» очередной панели инструментов:

```
procedure StickControls; virtual;
```

Этот метод нужно перекрыть, если стандартное размещение панелей по каким-либо причинам вас не устраивает.

TApplicationEvents — обработчик сообщений Windows

В отличие от остальных компонентов вкладки Additional компонент TApplicationEvents не имеет видимого эквивалента в работающей программе. Его основное и единственное назначение — служить приемником многочисленных сообщений, которые Windows посылает работающей программе.

Лишь очень немногие программы нуждаются в специальной (отличной от установленной по умолчанию) обработке сообщений — примером могут служить программы, эмулирующие клавиатуру DOS или фильтрующие ввод пользователя (в учебных целях, например).

Следующая простая программа познакомит вас с примером использования компонента. В ней фильтруется ввод программы в компонент TEdit так, чтобы запретить появление в нем любых цифр.

1. На пустую форму поместите компоненты TEdit и TApplicationEvents.
2. Напишите следующий обработчик события OnMessage компонента TApplicationEvents:

```
procedure TForm1.ApplicationEvents1Message(var Msg:
    tagMSG; var Handled: Boolean);
begin
    Handled := Edit1.Focused and (Msg.Message=wm_Char)
                and (Msg.wParam>=48) and (Msg.wParam<=57)
end;
```

Если теперь запустить программу, ввод цифр в компонент Edit1 будет блокирован. В нашей простой программе есть единственный компонент, способный получать фокус ввода, поэтому проверка свойства Edit1.Focused в операторе при-

сваивания является излишней. Я ввел ее только для того, чтобы продемонстрировать, как программа может определить получателя клавиатурного ввода.

Любая форма может иметь сколько угодно компонентов `TApplicationEvents`. Одно и то же сообщение поступает последовательно в каждый из них до тех пор, пока это сообщение не будет обработано или пока оно не поступит всем. Порядок получения сообщений обратен порядку размещения компонентов на форме: вначале сообщение получит последний размещенный на форме компонент, затем предпоследний и т. д. Этот порядок можно изменить с помощью метода `Activate`. Запретить получение остальным компонентам уже обработанного сообщения можно с помощью метода `CancelDispatch`. Если в проекте имеется несколько форм, расположенные на них компоненты `TApplicationEvents` начинают получать сообщения только после того, как соответствующая форма становится активной. Работа всех компонентов `TApplicationEvents` будет блокирована, если в объекте `Application` определен обработчик события `OnMessage`.

В инспекторе объектов представлены только два свойства компонента: `Name` и `Tag`.

Методы компонента `TApplicationEvents` перечислены в табл. 16.14. Его события не представляют для программиста особого интереса.

Таблица 16.14. Методы компонента `TApplicationEvents`

Метод	Описание
procedure <code>Activate;</code>	Делает компонент первым получателем сообщений
procedure <code>CancelDispatch;</code>	Запрещает получение текущего сообщения остальным компонентам

TValueListEditor — специализированный редактор списков

Компонент `TValueListEditor` предназначен для редактирования списков, строки которых состоят из пар `Имя=Значение`. Такие списки широко используются в реестре Windows и файлах инициализации. Наконец, начиная с версии 5, Delphi по умолчанию в таком формате хранит файлы описания форм с расширением `DFM`. Если на пустую форму поместить компонент `TValueListEditor`, диалоговый ресурс `TOpenDialog` и кнопку `TButton`, то после загрузки в редактор файла `Source\Chap_4\Example\Example.dfm` получим экран, показанный на рис. 16.10.

Обработчик щелчка на кнопке имеет такой вид:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
    if OpenDialog1.Execute then
        ValueListEditor1.Strings.LoadFromFile(OpenDialog1.FileName)
end;
```

Непосредственным предком компонента `TValueListEditor` является класс `TCustomDrawGrid`, поэтому многие его свойства, методы и события совпадают с описанными выше в этой главе свойствами, методами и событиями компонента `TStringGrid`.

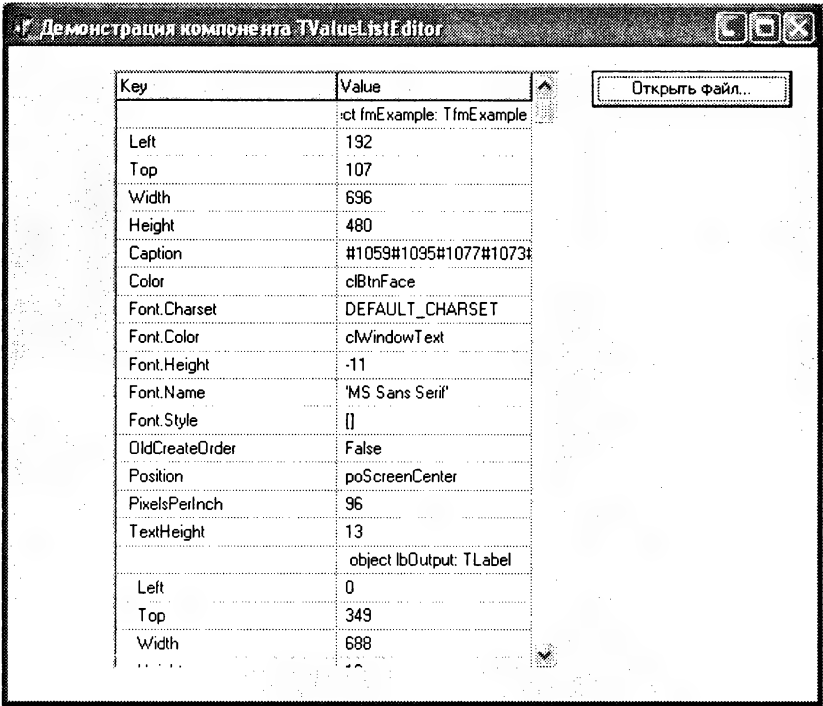


Рис. 16.10. Демонстрация компонента TValueListEditor

Специфичные свойства компонента TValueListEditor перечислены в табл. 16.15.

Таблица 16.15. Некоторые свойства компонента TValueListEditor

Свойство	Описание
property Cells[ACol, ARow: Integer]: String;	Открывает доступ к содержимому ячейки, лежащей на пересечении столбца ACol со строкой ARow (нумерация столбцов и строк начинается с нуля)
property ColCount: Integer;	Содержит количество столбцов
type TDisplayOption = (doColumnTitles, doAutoColResize, doKeyColFixed); TDisplayOptions = set of TDisplayOption; property DisplayOptions: TDisplayOptions;	Свойство может включать следующие значения: doColumnTitles — первая строка содержит заголовки столбцов, определенные свойством TitleCaption, и не прокручивается; doAutoColResize — предписывает автоматически изменять ширину столбцов при изменении размеров компонента; doKeyColFixed — если указано, ширина первого столбца, содержащего имена, не меняется при изменении размеров
property ItemProps [const KeyOrIndex: Variant]: TItemProp;	Сложное свойство, управляющее различными аспектами отображения значений (см. ниже)

— продолжение ➤

Таблица 16.15 (продолжение)

Свойство	Описание
type TKeyOption = (keyEdit, keyAdd, keyDelete, keyUnique); TKeyOptions = set of TDisplayOption; property KeyOptions: TKeyOptions;	Управляет возможным изменением списка: keyEdit — пользователь может изменять имя параметра (в первом столбце); keyAdd — пользователь может добавлять новую пару имя-значение нажатием клавиши Insert (требует включения keyEdit); keyDelete — пользователь может удалить выделенную пару нажатием клавиши Delete; keyUnique — названия вводимых пользователем имен должны быть уникальными
property Keys[Index: Integer]: String;	Открывает доступ к названиям значений по их индексам
property RowCount: Integer;	Содержит количество строк
property Strings: TStrings;	Содержит строки списка
property TitleCaptions: TStrings;	Содержит заголовки столбцов
property Values[const Key: String]: String;	Открывает доступ к значениям по их индексам

Свойство ItemProps класса TItemProp имеет собственные свойства (табл. 16.16).

Таблица 16.16. Свойства, относящиеся к свойству ItemProps класса TItemProp

Свойство	Описание
property EditMask: String;	Содержит маску для формирования значения. Правила создания маски описаны в разделе «TMaskEdit — поле с маской ввода» этой главы
type TEditStyle = (esSimple, esEllipsis, esPickList); property EditStyle: TEditStyle;	Определяет правила редактирования значения (см. ниже)
property MaxLength: Integer;	Определяет максимальную длину (в символах) значения. Если 0, нет ограничений на длину
property PickList: TStrings;	Содержит раскрывающийся список возможных значений
property ReadOnly: Boolean;	Разрешает/запрещает редактирование значения

Значения свойства EditStyle:

- ✎ esSimple — значение редактируется прямо в строке как в обычном однострочном текстовом поле;
- ✎ esEllipsis — в строку вставляется кнопка для вызова диалогового окна редактора значения (возбуждается событие OnEditButtonClick);

- ✱ **esPickList** — в строку вставляется кнопка для раскрытия списка возможных значений (возбуждается событие **OnGetPickList**).

Свойство **PickList** не предназначено для программного доступа. Чтобы создать раскрывающийся список возможных значений, нужно свойству **ItemProps.EditStyle** присвоить значение **esPickList** и написать обработчик события **OnGetPickList**. Пусть, например, первая отображаемая в компоненте строка должна иметь раскрывающийся список. Тогда необходим такой код:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
    ValueListEditor1.ItemProps[1].EditStyle := esPickList;
end;

procedure TForm1.ValueListEditor1GetPickList(Sender:
    TObject; const KeyName: String; Values: TStrings);
begin
    // Содержимое списка доступно через параметр Values:
    Values.Add('1');
    Values.Add('2');
    Values.Add('3');
    Values.Add('4');
end;
```

Методы компонента **TValueListEditor** перечислены в табл. 16.17.

Таблица 16.17. Методы компонента **TValueListEditor**

Метод	Описание
procedure DeleteRow (ARow: Integer); override ;	Удаляет из списка строку с индексом ARow. Свойство KeyOptions должно содержать флаг keyDelete
function FindRow(const KeyName: String ; var Row: Integer): Boolean;	Ищет в списке строку с именем KeyName и возвращает в параметре Row индекс найденной строки. Возвращает значение True в случае успеха поиска
function InsertRow(const KeyName, Value: String ; Append: Boolean): Integer;	Вставляет в список строку вида KeyName=Value. Если текущая строка пуста, вставляемая строка занимает ее место, в противном случае она занимает место перед текущей (если Append=False) или за ней (если Append=True). Возвращает индекс вставленной строки. Свойство KeyOptions должно содержать флаг keyAdd
procedure Refresh;	Перерисовывает компонент
function RestoreCurrentRow: Boolean;	Отменяет все изменения текущей строки и возвращает значение True в случае успеха

Методы **CellRect**, **MouseToCell** и **MouseCoord** компонента **TValueListEditor** аналогичны одноименным методам компонента **TStringGrid**.

TLabelEdit — однострочное поле с меткой

Компонент TLabelEdit представляет собой удобную комбинацию однострочного текстового поля с меткой. Надпись в метке указывается в свойстве EditLabel. Следующее свойство определяет положение метки относительно текстового поля:

```
type TLabelPosition = (lpAbove, lpBelow, lpLeft, lpRight);
property LabelPosition: TLabelPosition;
```

Расстояние от метки до текстового поля (в пикселах) определяется свойством LabelSpacing:

```
property LabelSpacing: Integer;
```

TColorBox — список выбора цвета

Компонент TColorBox представляет собой комбинированный список, пунктами которого являются цвета, то есть предназначен для отображения и выбора цвета. В отличие от компонентов TColorGrid (вкладка Samples) и TColorDialog (вкладка Dialogs), состав отображаемых им цветов может меняться.

Многие свойства, методы и события компонента совпадают с одноименными свойствами, методами и событиями его ближайшего «родственника» — компонента TComboBox.

Специфичные свойства компонента TColorBox перечислены в табл. 16.18.

Таблица 16.18. Некоторые свойства компонента TColorBox

Свойство	Описание
property ColorNames[Index: Integer]: String;	Возвращает имя цвета по его индексу в списке
property Colors[Index: Integer]: TColor;	Содержит список цветов
property DefaultColorColor: TColor;	Содержит цвет, заданный по умолчанию. Свойство Style должно включать флаги cbSystemColors и cbIncludeDefault
property NoneColorColor: TColor;	Определяет пункт без цвета. Свойство Style должно включать флаги cbSystemColors и cbIncludeNone
property Selected: TColor;	Определяет текущий выбранный цвет
type TColorBoxStyles = (cbStandardColors, cbExtendedColors, cbSystemColors, cbIncludeNone, cbIncludeDefault, cbCustomColor, cbPrettyNames); TColorBoxStyle = set of TColorBoxStyles;	Определяет отображаемые компонентом цвета: cbStandardColors — список содержит 16 основных цветов; cbExtendedColors — список содержит дополнительные цвета, определенные в модуле Graphics; cbSystemColors — в списке отображаются цвета clMoneyGreen, clSkyBlue, clCream и clMedGray; cbIncludeNone — список содержит пункт clNone; cbIncludeDefault — список содержит пункт clDefault;

Свойство	Описание
<code>property Style:</code> <code>TColorBoxStyle;</code>	<code>cbCustomColor</code> — первый пункт списка открывает стандартное диалоговое окно <code>TColorDialog</code> ; <code>cbPrettyNames</code> — имена цветов отображаются без префикса <code>c1</code>

TChart — диаграмма

Компонент TChart предназначен для графического представления числовых данных. На рис. 16.11 показан пример использования компонента.

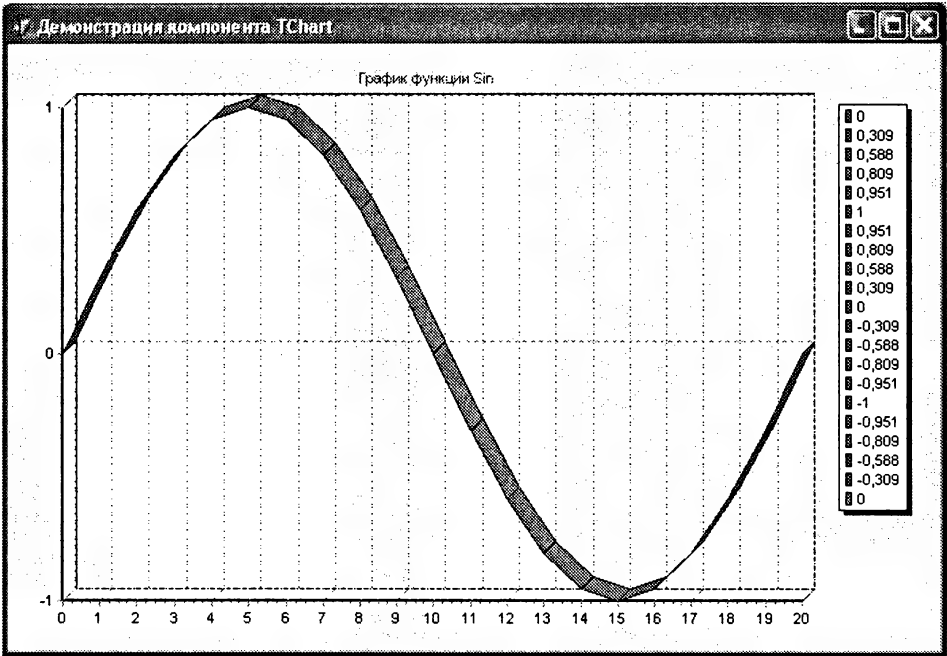


Рис. 16.11. Пример использования компонента TChart

Компонент содержит большое количество специфичных свойств, событий и методов — их детальное рассмотрение потребовало бы значительного объема книги и здесь не приводится.

Общая схема его использования выглядит следующим образом.

Вначале компонент помещается на форму. После щелчка на нем правой кнопкой мыши вызывается контекстное меню компонента, которое в числе прочих содержит команду `Edit Chart` — с помощью этой команды вызывается многостраничное окно редактора компонента. Вкладка `Series` этого окна открывает доступ к так называемым *сериям* — объектам класса `TChartSeries`, которые, собственно, и будут отображать наборы чисел в графическом виде. Чтобы отобразить данные, нужно создать как минимум одну серию — для этого следует в окне редактора щелкнуть на кнопке `Add` и выбрать подходящий тип диаграммы (рис. 16.12).

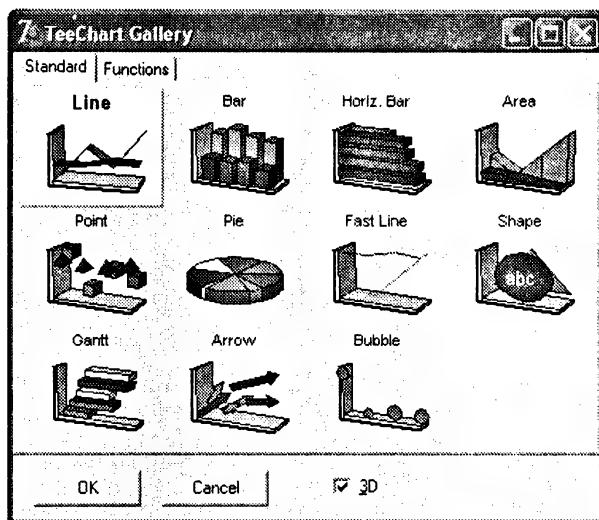


Рис. 16.12. Возможные виды диаграмм для компонента TChart

После закрытия окна редактора компонент будет содержать диаграмму, реальный вид которой зависит от фактических данных, генерируемых работающей программой и добавляемых к серии с помощью метода `AddX`, `AddY` или `AddXY` объекта `TChartSeries` (получить доступ к нужной серии можно с помощью индексированного свойства `SeriesList` — первая созданная серия имеет индекс 0, вторая 1 и т. д.). Например, график, показанный на рис. 16.11, создал следующий обработчик события `OnActivate` формы:

```
procedure TForm1.FormActivate(Sender: TObject);
var
  k: Integer;
begin
  for k := 0 to 20 do
    Chart1.SeriesList[0].AddXY(k, sin(k*pi/10), '', clRed);
end;
```

За более полной информацией о компоненте обратитесь к встроенной справочной службе Delphi.

TActionManager — менеджер действий

Вместе с тремя другими компонентами вкладки `Additional` — `TActionMainMenuBar`, `TActionToolBar` и `TCustomDlg` — компонент `TActionManager` позволяет создавать настраиваемый пользовательский интерфейс. Таким интерфейсом обладает, например, среда Delphi: вы можете добавить/удалить любую кнопку панели инструментов, запретить/разрешить выбор кнопок и команд меню и т. п.

Компонент `TActionManager` централизует однотипные действия, производимые схожими в функциональном отношении элементами интерфейса. Каждое такое действие компонент сохраняет в дисковом файле, что позволяет программе

«запоминать» последние настройки, сделанные пользователем. Таким образом, компонент значительно расширяет функциональность компонента TActionList (см. раздел «TActionList — список действий» в главе 15) и, по существу, делает его ненужным.

Компонент оперирует категориями однотипных действий (Edit, File, Format и т. п.) и может размещать категории в контейнере TActionMainMenuBar (в виде команд меню), а входящие в них действия — в контейнере TActionToolBar (в виде кнопок панели инструментов). В Delphi предусмотрены 12 стандартных категорий, многие действия которых заранее определены. Например, в категорию File входит предопределенное действие TFileExit. Выбор в меню команды или щелчок на кнопке панели инструментов, инициирующей это действие, приводит к завершению работы программы.

Познакомимся с методикой использования компонента на следующем примере (Source\Chap_16\ActionManager\ActMenDemo.dpr).

1. Поместите на пустую форму компоненты TActionManager, TActionMainMenuBar, TActionToolBar и TCustomizedDlg. По умолчанию контейнеры TActionMainMenuBar и TActionToolBar займут верхнюю часть формы, положение двух других компонентов не имеет значения.
2. Дважды щелкните на компоненте или вызовите его контекстное меню и выберите команду Customize — на экране появится окно редактора компонента. Перейдите на вкладку Actions (рис. 16.13).

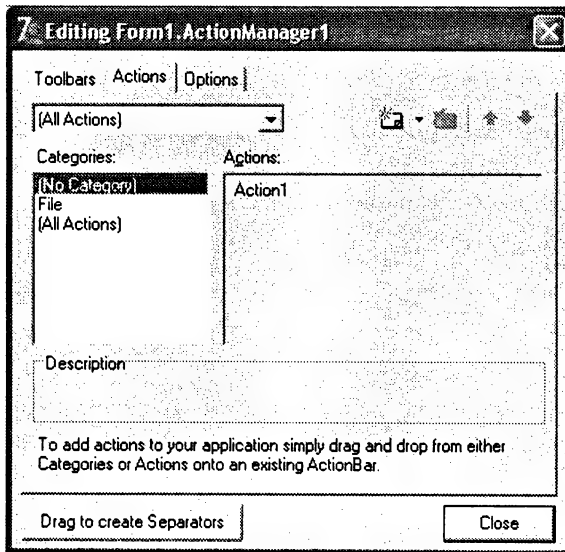


Рис. 16.13. Редактор компонента TActionManager

3. Щелкните в списке Actions правой кнопкой мыши и выберите в контекстном меню команду New Standard Action или просто нажмите клавиши Ctrl+Ins — на экране появится дополнительное окно с перечнем всех определенных в Delphi классов для стандартных действий (рис. 16.14).

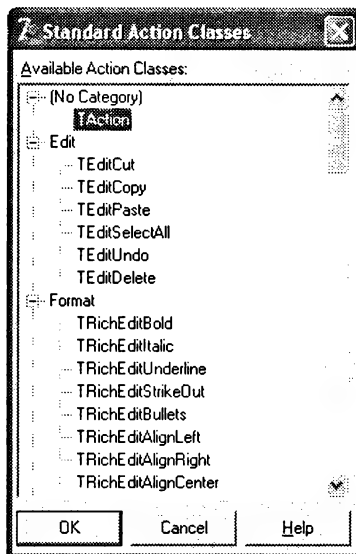


Рис. 16.14. Окно со списком стандартных действий

4. С помощью полосы прокрутки выберите в списке все классы, связанные с категорией File (для выбора нескольких классов одновременно щелкайте на каждом, удерживая нажатой клавишу Ctrl) и закройте окно щелчком на кнопке ОК.
5. «Схватите» мышью категорию File в списке Categories (см. рис. 16.13) и перетащите ее на компонент TActionMainMenuBar. Из списка Action перетащите на компонент TActionToolBar элемент Exit.
6. Поместите в свойство/событие ActionManager компонента TCustomizedDlg ссылку на компонент ActionManager1.
7. Для формы напишите следующий обработчик события OnDb1Click:

```
procedure TForm1.FormDb1Click(Sender: TObject);
begin
    CustomizeDlg1.Show
end;
```

Если вы запустите программу, то увидите пустое окно с меню File и кнопкой Exit на панели инструментов, причем команды меню (кроме недоступной команды Run) и кнопка на панели инструментов будут выполнять ожидаемые от них действия. Двойной щелчок на форме активизирует уже знакомое нам (см. рис. 16.13) окно редактора ActionManager1, и вы сможете изменять состав команд меню и кнопок панели инструментов.

Глава 17

Компоненты вкладки Win32

Компоненты вкладки Win32 предназначены для разработки программ с усложненным интерфейсом — предоставляемые ими элементы управления в традиционных программах обычно не требуются.

TTabControl — набор вкладок

Компонент TTabControl представляет собой контейнер с вкладками. Свойство Tabs определяет названия и количество вкладок. Событие OnChange возникает при выборе новой вкладки и позволяет управлять содержимым окна компонента. Для примера на рис. 17.1 показано использование компонента для управления календарем (с помощью компонента TTabControl созданы вкладки с названиями месяцев).

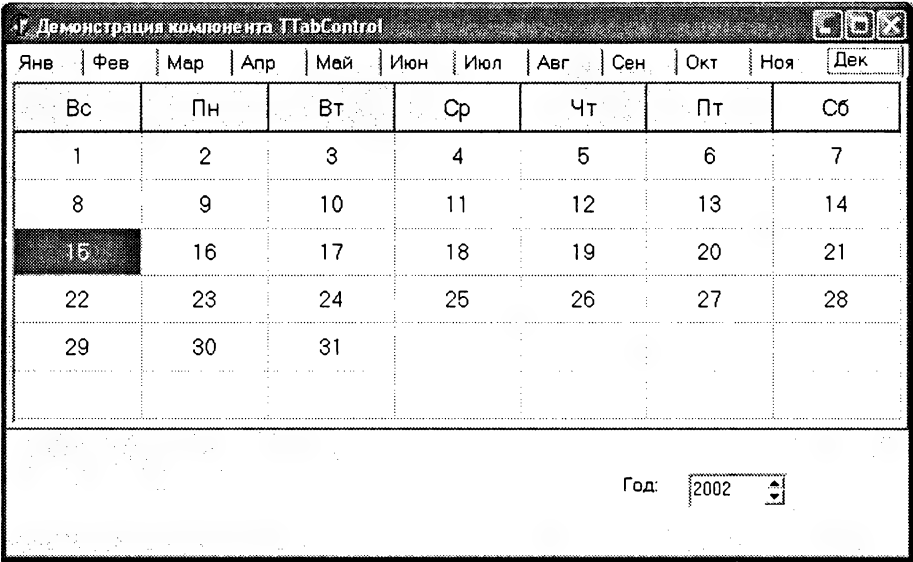


Рис. 17.1. Управление календарем с помощью компонента TTabControl

Свойства компонента TTabControl представлены в табл. 17.1.

Таблица 17.1. Свойства компонента TTabControl

Свойство	Описание
property DisplayRect: TRect;	Определяет рабочую зону компонента, предназначенную для размещения других компонентов. Клиентская часть компонента содержит зону вкладок и рабочую зону
property HotTrack: Boolean;	Если содержит значение True, название вкладки автоматически выделяется цветом, когда на ней находится указатель мыши
property MultiLine: Boolean;	Разрешает располагать вкладки в несколько рядов. Если содержит значение False, и вкладки не умещаются в границах компонента, в зону вкладок автоматически вставляются кнопки прокрутки
property ScrollOpposite: Boolean;	Разрешает/запрещает перемещение неактивных рядов вкладок на противоположную сторону компонента. Учитывается, если количество рядов больше 2
property TabHeight: Smallint;	Определяет высоту каждой вкладки в пикселах. Если содержит 0, высота вкладок выбирается автоматически в зависимости от выбранного шрифта
property TabIndex: Integer;	Определяет индекс выбранной вкладки или содержит -1, если ни одна из вкладок не выбрана. Индексация начинается с 0
TTabPosition=(tpTop, tpBottom); property TabPosition: TTabPosition;	Определяет положение зоны вкладок относительно рабочей зоны компонента (tpTop — сверху, tpBottom — внизу)
property Tabs: TStrings;	Определяет надписи на вкладках и их количество. Чтобы добавить или удалить вкладку, нужно добавить ее надпись в список Tabs или удалить надпись из списка
property TabWidth:	Определяет ширину каждой вкладки в пикселах. Если содержит 0, ширина каждой вкладки выбирается индивидуально, в зависимости от длины ее надписи

Помимо события OnChange, возникающего после перехода к новой вкладке, для компонента определено также событие OnChanging, которое возникает перед сменой вкладки:

```

type TTabChangingEvent = procedure (Sender: TObject;
                                     var AllowChange: Boolean) of
object;
property OnChanging: TTabChangingEvent;

```

Обработчик события может запретить выбор вкладки, вернув в параметре AllowChange значение False.

TPageControl — набор страниц с вкладками

Компонент TPageControl в отличие от компонента TTabControl может содержать несколько перекрывающихся друг друга страниц (панелей) класса TTabSheet. Каждая страница выбирается щелчком на связанной с ней вкладке и может содержать свой набор помещенных на нее компонентов.

Чтобы на этапе конструирования добавить новую страницу или выбрать ранее вставленную, щелкните на компоненте правой кнопкой мыши и выберите в контекстном меню команду **New Page** (Новая страница), **Next Page** (Следующая страница) или **Previous Page** (Предыдущая страница). Смена страниц идет циклически, то есть после показа последней отображается первая и наоборот.

Помимо свойств HotTrack, MultiLine, ScrollOpposite, TabHeight, TabPosition и TabWidth, которые аналогичны одноименным свойствам компонента TTabControl, компонент TPageControl имеет специфические свойства, перечисленные в табл. 17.2.

Таблица 17.2. Свойства компонента TPageControl

Свойство	Описание
property ActivePage: TTabSheet;	Содержит ссылку на активную страницу. Установка нового значения ActivePage размещает соответствующую страницу поверх остальных. Для выбора новой страницы следует использовать методы SelectNextPage и FindNextPage
property PageCount: Integer;	Содержит количество страниц (только для чтения)
property Pages [Index: Integer]: TTabSheet;	Возвращает ссылку на страницу по ее индексу (только для чтения)

Методы компонента TPageControl перечислены в табл. 17.3.

Таблица 17.3. Методы компонента TPageControl

Метод	Описание
function FindNextPage (CurPage: TTabSheet; GoForward, CheckTabVisible: Boolean): TTabSheet;	Ищет следующую страницу: CurPage — текущая страница; GoForward — содержит значение True, если поиск идет от первой страницы к последней; CheckTabVisible — содержит значение True, если из поиска исключаются страницы с признаком TabVisible=False. Возвращает ссылку на найденную страницу. Если страница CurPage не принадлежит компоненту, возвращает ссылку на первую или последнюю страницу в зависимости от параметра GoForward
procedure SelectNextPage (GoForward: Boolean);	Делает активной другую страницу: если GoForward=True, активизируется следующая страница, в противном случае — предыдущая

TImageList — хранилище изображений

Компонент класса TImageList представляет собой контейнер для хранения множества рисунков одинакового размера. Он может быть полезен при программном создании набора кнопок для панели инструментов, секций управляющего заголовка, анимационных эффектов и вообще там, где требуется индексированный доступ к изображениям.

Компонент не имеет собственной канвы и поэтому не может самостоятельно отображать хранимые в нем изображения, которые могут быть растрами (BMP) или значками (ICO). Его метод Draw получает канву от другого компонента и рисует изображение в клиентской области этого компонента. В компоненте есть множество специальных свойств и методов, упрощающих обслуживание набора изображений. Для получения дополнительной информации обратитесь к встроенной справочной службе. Пример использования компонента имеется в разделе «TControlBar — контейнер для панели инструментов» главы 16.

TRichEdit — поле формата RTF

Компонент TRichEdit представляет собой многострочное редактируемое текстовое поле, работающее с форматом RTF (Rich Text Format — расширенный текстовый формат). Текст формата RTF хранит дополнительную служебную информацию, управляющую свойствами каждого абзаца и сменой шрифта по ходу текста.

На рис. 17.2 показан пример использования компонента. Полный текст программы, с помощью которой создан рисунок, приводится в проекте Source\Chap_17\RichEditDemo\RichEdDemo.dpr.

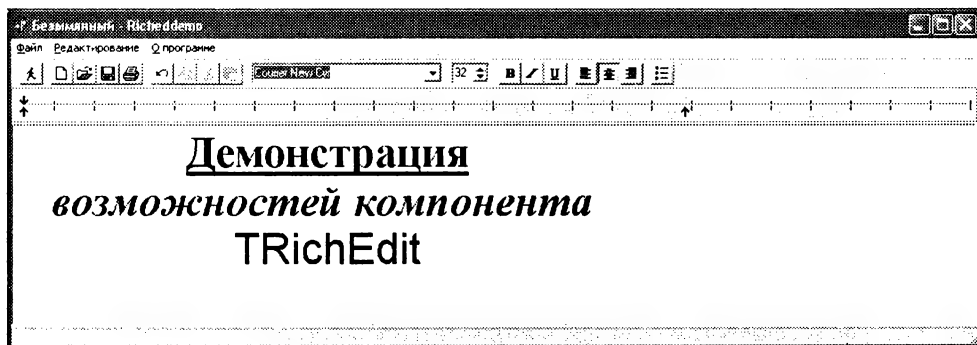


Рис. 17.2. Окно демонстрационной программы RichEdDemo

Значительный объем кода программы и достаточно сложный ее интерфейс не позволяют вставить в книгу листинг программы и описание процесса конструирования формы — смотрите проект, снабженный подробными комментариями. Отмечу несколько любопытных деталей. Прежде всего обратите внимание на значки, определяющие границы абзаца. Они реализованы с помощью меток TLabel1, использующих шрифт Windings. Набор символов этого шрифта представляет собой множество разного рода значков, заменяющих стандартное начертание того или иного символа. В частности, кириллические символы «й» и «к» заменяются соот-

ветственно значками  и . Чтобы перемещение мышью этих значков на новое место приводило к нужному изменению той или иной границы, необходимо положение метки записать в свойство XXXIndent (FirstIndent, LeftIndent, RightIndent — см. ниже) основного или выделенного текста. Обратите внимание: при установке границы компонент учитывает значение соответствующего свойства с коэффициентом 4/3. Поэтому истинное положение метки перед установкой в свойство умножается на 0,75.

Компонент использует вспомогательные объекты класса TTextAttributes для хранения атрибутов шрифта. Эти атрибуты распространяются на весь текст через свойство компонента DefAttributes или на выделенную часть текста — через его свойство SelAttributes. Помимо обычных шрифтовых свойств CharSet, Color, Height, Name, Pitch, Size и Style (см. класс TFont) объект TTextAttributes имеет также свойства ConsistentAttributes и Protected. Первое доступно только для чтения и содержит набор текстовых характеристик, общих как для всего текста, так и для его выделенной части. Свойство Protected защищает весь текст или его части от редактирования. Попытка изменить текст, имеющий атрибут Protected, вызывает обработчик события OnProtectChange, который может разрешить или запретить изменения. По умолчанию изменения запрещены.

Для каждого текстового абзаца создается объект класса TParaAttributes, в котором сохраняются атрибуты абзаца. Эти атрибуты доступны через свойства класса TParaAttributes, перечисленные в табл. 17.4.

Таблица 17.4. Свойства класса TParaAttributes

Свойство	Описание
property Alignment: TAlignment;	Определяет горизонтальное выравнивание текста абзаца относительно границ компонента
property FirstIndent: LongInt;	Определяет отступ текста первой строки абзаца в пикселах от левого края рабочей зоны компонента
property LeftIndent: LongInt;	Определяет отступ остальных строк текста абзаца в пикселах от границы, устанавливаемой свойством FirstIndent
TNumberingStyle = (nsNone, nsBullet); property Numbering: TNumberingStyle;	Определяет, надо ли вставлять слева от абзаца маркеры списка. Если содержит значение nsBullet, маркеры списка вставляются
property RightIndent: LongInt;	Определяет отступ текста абзаца в пикселах от правого края компонента
property Tab[Index: Byte]: LongInt;	Для табулостопов с индексом Index содержит его позицию в пикселах от левого края компонента
property TabCount: Integer;	Определяет количество табулостопов в строке абзаца

ПРИМЕЧАНИЕ По поводу значений свойств XXXIndent см. выше пояснение к демонстрационному примеру.

В табл. 17.5 представлены свойства компонента TRichEdit.

Таблица 17.5. Свойства компонента TRichEdit

Свойство	Описание
property DefAttributes: TTextAttributes;	Определяет шрифтовые атрибуты всего текста
property HideScrollBars: Boolean;	Определяет, будут ли автоматически появляться полосы прокрутки, если текст отсекается границами компонента. Игнорируется, если свойство ScrollBars содержит значение ssNone
property HideSelection: Boolean;	Определяет, будет ли убираться выделение текста, если компонент потеряет фокус ввода
property Lines: TStrings;	Содержит набор строк текста. С помощью его методов LoadFromFile и SaveToFile компонент может читать текст из файла или записывать в него текст
property PageRect: TRect;	Определяет размеры страницы при печати на принтере
property Paragraph: TParaAttributes;	Содержит атрибуты текущего абзаца, то есть абзаца с выделением или с текстовым курсором. Программа не может изменить свойство Paragraph, но может изменить свойства связанного с ним абзаца
property PlainText: Boolean;	Запрещает/разрешает записывать в файл или читать из него служебную информацию формата RTF (True — запрещает)
property SelAttributes: TTextAttributes;	Определяет шрифтовые атрибуты выделенного текста
property SelLength: Integer;	Задаёт длину в символах выделенной части текста
property SelStart: Integer;	Определяет номер первого символа выделенной части текста от начала текста (нумерация символов начинается с 0). Если нет выделения, определяет символ, перед которым располагается текстовый курсор
property SelText: String;	Содержит выделенный текст. Установка нового значения SelText замсняет выделенный текст новым, а если нет выделения — вставляет его в позицию курсора

Свойство Text компонента, похоже, не имеет ограничений — во всяком случае, мне удалось разместить в нем 800 000 байт (проект Chap_17\RichEditTest\RDTest.dpr). Замечу, что метод Lines.Add выполняется в нем намного медленнее, чем в компоненте TМемо, так что ждать окончания размещения в компоненте 50 000 строк по 16 байт каждая мне пришлось около 20 минут (1700 МГц, 256 Мбайт, Windows XP).

Методы компонента TRichEdit перечислены в табл. 17.6.

Таблица 17.6. Методы компонента TRichEdit

Метод	Описание
procedure Clear;	Удаляет весь текст
TSearchType = (stWholeWord, stMatchCase); TSearchTypes = set of TSearchType; function FindText(const SearchStr: String ; StartPos, Length: Integer; Options: TSearchTypes): Integer;	Ищет в тексте строку SearchStr и возвращает индекс первого ее символа при удачном поиске: StartPos — начало поиска; Length — длина строки. Options указывает, будет ли поиск идти по целым словам и надо ли учитывать регистр букв
function GetSelTextBuf(Buffer: PChar; BufSize: Integer): Integer;	Копирует не более BufSize символов выделенного текста в буфер Buffer и возвращает количество скопированных символов
procedure Print(const Caption: String);	Форматирует текст по границам листа бумаги и печатает его на принтере, заданном по умолчанию. Параметр Caption определяет заголовок печати

События компонента TRichEdit представлены в табл. 17.7.

Таблица 17.7. События компонента TRichEdit

Событие	Описание
TRichEditProtectChange = procedure (Sender: TObject; StartPos, EndPos: Integer; var AllowChange: Boolean) of object ; property OnProtectChange: TRichEditProtectChange;	Возникает при попытке изменить текст, имеющий атрибут Protected: StartPos, EndPos — соответственно начальная и конечная позиции изменяемого текста. В параметре AllowChange обработчик возвращает значение True, если можно изменять текст
TRichEditResizeEvent = procedure (Sender: TObject; Rect: TRect) of object ; property OnResizeRequest: TRichEditResizeEvent;	Событие связано с перерисовкой текста из-за изменения размеров шрифта. Параметр Rect содержит прямоугольник, который будет перерисован
TRichEditSaveClipboard = procedure (Sender: TObject; NumObjects, NumChars: Integer; var SaveClipboard: Boolean) of object ; property OnSaveClipboard: TRichEditSaveClipboard;	Возникает в момент удаления компонента и извещает его о том, что в буфере обмена осталась помещенная компонентом информация. Если обработчик вернет значение False в параметре SaveClipboard, буфер обмена будет очищен
property OnSelectionChange: TNotifyEvent;	Возникает при изменении выделенного текста

TTrackBar — ползунок

Компонент класса TTrackBar предназначен для плавного изменения числовой величины. Он во многом схож с элементом управления TScrollBar (см. раздел «TScrollBar — полоса прокрутки» в главе 15) и отличается от него, в основном, оформлением.

Свойства компонента TTrackBar представлены в табл. 17.8.

Таблица 17.8. Свойства компонента TTrackBar

Свойство	Описание
property Frequency: Integer;	Определяет частоту нанесения меток: 1 — каждое значение диапазона изменения имеет метку; 2 — каждое 2-е значение имеет метку и т. д.
property LineSize: Integer;	Определяет минимальное смещение ползунка при нажатии клавиш перемещения курсора или перетаскивании мышью
property Max: Integer;	Определяет максимальное значение диапазона изменения
property Min: Integer;	Определяет минимальное значение диапазона изменения
property Orientation: TTrackBarOrientation;	Определяет ориентацию компонента: trHorizontal — горизонтальная; trVertical — вертикальная
property PageSize: Integer;	Определяет смещение ползунка при нажатии клавиш Page Up, Page Down или при щелчке мышью на концах шкалы
property Position: Integer;	Определяет текущее положение ползунка
property SelEnd: Integer;	Задаёт конечную позицию выделения
property SelStart: Integer;	Задаёт начальную позицию выделения
TTickMark = (tmBottomRight, tmTopLeft, tmBoth); property TickMarks: TTickMark;	Определяет способ нанесения меток: tmBottomRight — внизу или справа; tmTopLeft — вверху или слева; tmBottom — по обоим сторонам
TTickStyle = (tsNone, tsAuto, tsManual); property TickStyle: TTickStyle;	Определяет стиль нанесения меток: tsNone — нет меток; tsAuto — метки наносятся с частотой Frequency; tsManual — наносятся начальная и конечная метки, остальные наносит программа с помощью метода SetTick

С помощью следующего метода метка устанавливается в позицию, определяемую значением Value:

procedure SetTick(Value: Integer);

TProgressBar — индикатор процесса

Компонент TProgressBar предназначен для отображения хода выполнения длительного по времени процесса. Он имеет много общего с компонентами TScrollBar

и TTrackBar, но в отличие от них у него нет регулятора, то есть с его помощью можно только отображать числовую величину, но пользователь не может ее изменять.

В табл. 17.9 представлены свойства, а в табл. 17.10 — методы компонента TProgressBar.

Таблица 17.9. Свойства компонента TProgressBar

Свойство	Описание
property Max: Integer;	Определяет максимальное значение диапазона изменения свойства Position
property Min: Integer;	Определяет минимальное значение диапазона изменения свойства Position
property Position: Integer;	Содержит текущее значение отображаемой величины
property Step: Integer;	Шаг наращивания свойства Position методом StepIt

Таблица 17.10. Методы компонента TProgressBar

Метод	Описание
procedure StepBy (Delta: Integer);	Наращивает значение свойства Position на величину Delta
procedure StepIt;	Наращивает значение свойства Position на величину, определяемую свойством Step

TUpDown — счетчик

Компонент TUpDown предназначен для пошагового регулирования числовой величины. Он имеет пару кнопок, с помощью которых величина наращивается или уменьшается. Обычно компонент TUpDown связан с компонентом класса TEdit (текстовым полем), который отображает регулируемую величину и при необходимости обеспечивает возможность ее непосредственного редактирования. Связанный компонент называется компаньоном.

Свойства компонента TUpDown перечислены в табл. 17.11.

Таблица 17.11. Свойства компонента TUpDown

Свойство	Описание
TUDAlignButton = (udLeft, udRight); property AlignButton: TUDAlignButton;	Определяет положение счетчика относительно компаньона: udLeft — счетчик располагается слева от него; udRight — справа от него
property ArrowKeys: Boolean;	Разрешает/запрещает интерпретировать нажатие клавиш ↑ и ↓ как щелчки, соответственно на направленной вверх и направленной вниз кнопках счетчика. Игнорируется, если компонент не связан свойством Associate с компаньоном

— продолжение ➤

Таблица 17.11 (продолжение)

Свойство	Описание
property Associate: TWinControl;	Определяет связанный компонент (компаньон). Кнопки автоматически располагаются с пужной стороны компаньона и выравнивают свою высоту по его высоте
property Increment: SmallInt;	Определяет шаг наращивания/уменьшения регулируемой величины
property Max: SmallInt;	Определяет максимальное значение диапазона изменения регулируемой величины
property Min: SmallInt;	Определяет минимальное значение диапазона изменения регулируемой величины
TUDOrientation = (udHorizontal, udVertical); property Orientation: TUDOrientation;	Определяет ориентацию компонента: udHorizontal — по горизонтали; udVertical — по вертикали
property Position: SmallInt;	Содержит текущее значение регулируемой величины
property Thousands: Boolean;	Если содержит значение True, при отображении числовой величины в компаньоне в текст вставляются разделители тысяч
property Wrap: Boolean;	Запрещает/разрешает выход значения Position из диапазона Max...Min (True — запрещает)

Для компонента определены два события. Первое событие:

```

type TUDChangingEvent = procedure (Sender: TObject;
var AllowChange: Boolean) of object;
property OnChanging: TUDChanginEvent;

```

Это событие возникает при любом изменении регулируемой величины. Обработчик события в параметре AllowChange сообщает, может ли величина измениться. Второе событие:

```

type TUDBtnType = (btNext, btPrev);
type TUDClickEvent = procedure (Sender: TObject;
                                Button: TUDBtnType);
property OnClick: TUDClickEvent;

```

Это событие возникает при щелчке на кнопках элемента. Параметр Button определяет нажатую кнопку: btPrev — вниз или влево; btNext — вверх или вправо.

TAnimate — анимация

Компонент TAnimate представляет собой проигрыватель видеоклипов формата AVI (Audio Video Interleaved — чередование аудио и видео). Компонент воспроизводит визуальную часть файла AVI и игнорирует его звуковое сопровождение. Он способен показывать лишь несжатое изображение или изображение, сжатое по

методу RLE (Run-Length Encoding). Изображение воспроизводится в отдельном потоке команд, что освобождает ресурсы программы для выполнения необходимой работы на фоне воспроизведения клипа.

В проекте Chap_17\TAnimate\AnimateDemo.dpr вы сможете просмотреть несколько коротких AVI-файлов (рис. 17.3). В листинге 17.1 показан обработчик события OnClick, с помощью которого выбирается и воспроизводится нужный файл.



Рис. 17.3. Демонстрация компонента TAnimate

Листинг 17.1. Обработчики события OnClick кнопки¹

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var
  S: String;
begin
  GetDir(0,S);
  OpenDialog1.InitialDir := S;
  if OpenDialog1.Execute then
    // Указываем файл с клипом
    Animatel.FileName := OpenDialog1.FileName;
    Animatel.Active := True;           // Запускаем клип
end;
```

Свойства компонента TAnimate представлены в табл. 17.12.

Таблица 17.12. Свойства компонента TAnimate

Свойство	Описание
property Active: Boolean;	Разрешает/запрещает демонстрацию клипа. Во время демонстрации содержит значение True
property AutoSize: Boolean;	Если содержит значение True, компонент автоматически устанавливает свои размеры так, чтобы полностью вместить изображение кадра

продолжение ↗

¹ Все представленные в книге тексты программ можно загрузить с сайта издательства по адресу <http://www.piter.com/download> — Примеч. ред.

Таблица 17.12 (продолжение)

Свойство	Описание
property Center: Boolean;	Центрирует изображение кадра в границах компонента по горизонтали и по вертикали
TCommonAVI = (aviNone, aviFindFolder, aviFindFile, aviFindComputer, aviCopyFiles, aviCopyFile, aviRecycleFile, aviEmptyRecycle, aviDeleteFile); property CommonAVI: TCommonAVI;	Задаёт один из стандартных видеоклипов, входящих в библиотеку SHELL32.DLL (см. рис. 17.4)
property FileName: TFileName;	Связывает компонент с AVI-файлом
property FrameCount: Integer;	Содержит количество кадров, показанных с начала демонстрации клипа
property FrameHeight: Integer;	Высота в пикселах одного кадра клипа
property FrameWidth: Integer;	Ширина кадра
property Open: Boolean;	Содержит значение True, если компонент связан с AVI-клипом и готов к работе
property Repetitions: Integer;	Определяет количество повторений клипа. Если 0, клип повторяется до тех пор, пока свойство Active содержит значение True
property ResHandle: THandle;	Определяет дескриптор ресурсного файла, содержащего AVI-клип
property ResHandle: THandle;	Определяет идентификатор ресурса с клипом в ресурсном файле
property ResName: String;	Определяет имя ресурса с клипом в ресурсном файле
property StartFrame: SmallInt;	Содержит номер начального кадра демонстрации (нумерация кадров начинается с 1)
property StopFrame: SmallInt;	Содержит номер конечного кадра демонстрации (нумерация кадров начинается с 1)
property Timers: Boolean;	Разрешает/запрещает синхронизацию по таймеру. Если содержит значение True, демонстрация синхронизируется сигналами таймера. Если содержит значение False, для демонстрации используется независимый поток команд
property Transparent: Boolean;	Если содержит значение True, фон клипа не накладывается на фон компонента

С помощью свойства CommonAVI можно заставить компонент показывать один из стандартных видеоклипов, представленных на рис. 17.4.

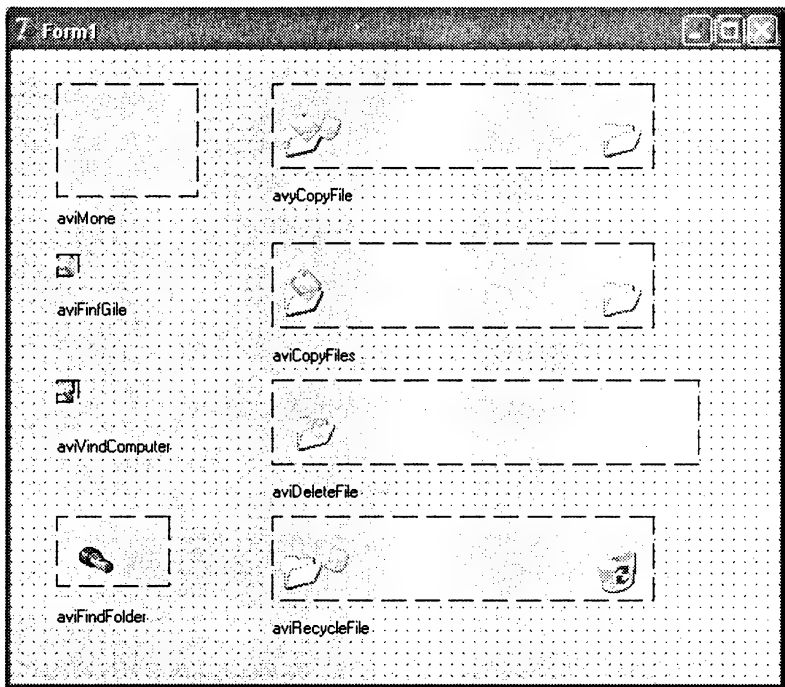


Рис. 17.4. Стандартные видеоклипы, определяемые свойством CommonAVI

Методы компонента TAnimate перечислены в табл. 17.13.

Таблица 17.13. Методы компонента TAnimate

Метод	Описание
procedure Play(FromFrame, ToFrame: Word; Count: Integer);	Демонстрирует Count раз подряд фрагмент клипа, начиная с кадра FromFrame по ToFrame включительно (нумерация кадров начинается с 1)
procedure Reset;	Восстанавливает исходное состояние компонента. Свойство Open вновь становится равным значению True, но свойство Active=False
procedure Seek(Frame: SmallInt);	Пропускает (не показывает) кадр с номером Frame (нумерация кадров начинается с 1)
procedure Stop;	Прекращает показ клипа

События компонента TAnimate перечислены в табл. 17.14.

Таблица 17.14. События компонента TAnimate

Событие	Описание
property OnClose: TNotifyEvent;	Возникает при установке значения False в свойство Open. Например, когда компонент демонстрирует несколько видеоклипов подряд

— продолжение ➤

Таблица 17.14 (продолжение)

Событие	Описание
property OnOpen: TNotifyEvent;	Возникает при установке значения True в свойство Open
property OnStart: TNotifyEvent;	Возникает в момент начала демонстрации
property OnStop: TNotifyEvent;	Возникает в момент прекращения демонстрации

TDateTimePicker — ввод и отображение даты/времени

Компонент предназначен для ввода и/или отображения даты и времени. На рис. 17.5 показаны различные варианты использования компонента.

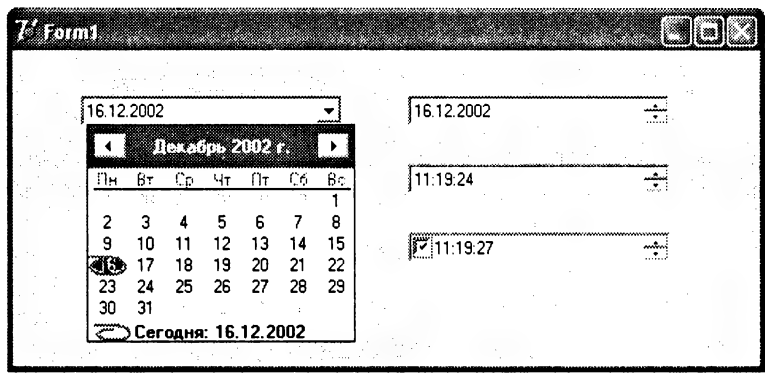


Рис. 17.5. Примеры использования компонента TDateTimePicker

При показе даты в режиме `DateMode=dmComboBox` календарь можно не раскрывать и установить дату вручную в верхнем окне. Это удобно, если нужно установить сразу и день, и месяц, и год. На раскрытом календаре изменить месяц можно с помощью небольших кнопок в верхней части компонента. Календарь закрывается после выбора даты или при щелчке на раскрывающей кнопке. Установив в свойство `Kind` значение `dtkTime`, можно заставить компонент отображать время. Это время в момент установки компонента на форму соответствует системному времени, и в дальнейшем его можно изменить, задав новое значение свойству `Time`.

Свойства компонента `TDateTimePicker` перечислены в табл. 17.15.

Таблица 17.15. Свойства компонента TDateTimePicker

Свойство	Описание
<code>TDTCalAlignment = (dtaLeft, dtaRight);</code> property CalAlignment: <code>TDTCalAlignment;</code>	Определяет положение раскрывающегося календаря: <code>dtaLeft</code> — слева от компонента; <code>dtaRight</code> — справа от компонента. Учитывается только для <code>Kind=dtkDate</code> и <code>DateMode=dmComboBox</code>

Свойство	Описание
property CalColors: TDateTimeColors;	С календарем связан объект класса TDateTimeColors, свойства которого определяют цвета календаря
property Checked: Boolean;	Если выполняются условия Checked=True и ShowCheckBox=True, флажок рядом с полем даты (времени) будет показан и установлен. Наличие флажка или его отсутствие разрешает или запрещает ручной ввод даты или времени
property Date: TDate;	Содержит введенную дату
TDTDateFormat = (dfShort, dfLong); property DateFormat: TDTDateFormat;	Определяет формат показа даты: dfShort — 17.09.99; dfLong — 8 Сентября 1999 г.
TDTDateMode = (dmComboBox, dmUpDown); property DateMode: TDTDateMode;	Определяет способ выбора даты: dmComboBox — с помощью раскрывающегося календаря; dmUpDown — с помощью встроенного компонента TUpDown
TDateTimeKind = (dtkDate, dtkTime); property Kind: TDateTimeKind;	Определяет содержимое компонента: dtkDate — дата; dtkTime — время
property MaxDate: TDate;	Определяет максимальную дату, которую может выбрать или ввести пользователь
property MinDate: TDate;	Определяет минимальную дату, которую может выбрать или ввести пользователь
property ParseInput: Boolean;	Если содержит значение True, возникает событие OnUserInput при каждом вводе пользователя в текстовое поле
property ShowCheckBox: Boolean;	Если содержит значение True, рядом с полем даты (времени) вставляется флажок, с помощью которого пользователь может запретить ручное изменение даты (времени)
property Time: TTime;	Содержит введенное пользователем время

События компонента TDateTimePicker представлены в табл. 17.16.

Таблица 17.16. События компонента TDateTimePicker

Событие	Описание
property OnChange: TNotifyEvent;	Возникает при любом изменении содержимого компонента
property OnCloseUp: TNotifyEvent;	Возникает при закрытии календаря
property OnDropDown: TNotifyEvent;	Возникает при открытии календаря

продолжение ⇨

Таблица 17.16 (продолжение)

Событие	Описание
DTParseInputEvent = procedure (Sender: TObject; const UserString: String ; var DateAndTime: TDateTime; var AllowChange: Boolean) of object ; property OnUserInput: TDTParseInputEvent;	Возникает при ручном вводе в текстовое поле, если ParseInput=True. UserString — введенная пользователем строка; DateAndTime — значения свойств Date и Time. В параметре AllowChange обработчик разрешает или запрещает изменение даты или времени

TMonthCalendar — календарь

Компонент предназначен для выбора или отображения даты. Он очень похож на календарь, который появляется в компоненте TDateTimePicker (при значении dmComboBox в свойстве DateMode), но в отличие от последнего может отображать одновременно несколько смежных месяцев — в зависимости от своих размеров.

Свойства компонента TMonthCalendar представлены в табл. 17.17.

Таблица 17.17. Свойства компонента TMonthCalendar

Свойство	Описание
property CalColors: TMonthCalColors;	С компонентом связан объект класса TMonthCalColors (см. ниже)
property Date: TDate;	Определяет выбранную дату
property EndDate: TDate;	Определяет конечную дату диапазона выбранных дат
type TCalDayOfWeek = (dowMonday, dowTuesday, dowWednesday, dowThursday, dowFriday, dowSaturday, dowSunday, dowLocaleDefault); property FirstDayOfWeek: TCalDayOfWeek;	Определяет первый день недели
property MaxDate: TDate;	Содержит максимальную дату, месяц которой еще будет доступен для отображения в компоненте. Если свойство содержит пустое значение, отображается любой следующий месяц
property MaxSelectRange: Integer;	Содержит максимальное количество дат в выбранном диапазоне
property MinDate: TDate;	Содержит минимальную дату, месяц которой еще будет доступен для отображения в компоненте. Если свойство содержит пустое значение, отображается любой предыдущий месяц

Свойство	Описание
property MultiSelect: Boolean;	Разрешает/запрещает возможность выделения диапазона дат
property ShowToday: Boolean;	Разрешает/запрещает показ текущей даты (по показаниям системных часов) внизу календаря
property ShowTodayCircle: Boolean;	Разрешает/запрещает обводить кружком текущую дату
property WeekNumbers: Boolean;	Разрешает/запрещает показ порядковых номеров недель от начала года в левой колонке

Свойства объекта класса TMonthCalColors:

- BackColor — цвет фона, разделяющего смежные месяцы;
- MonthBackColor — цвет фона дат;
- TextColor — цвет дат;
- TitleBackColor — цвет фона заголовка месяца;
- TitleTextColor — цвет текста заголовка;
- TrailingTextColor — цвет текста ведущих и ведомых дат.

TTreeView — иерархическое дерево

Компонент TTreeView служит для показа ветвящихся иерархических структур, таких как дерево наследования объектов или файловая структура диска. Он содержит связанные узлы, каждый из которых может содержать значок, текст и произвольный объект. Любой узел может иметь собственный список дочерних узлов, которые можно раскрывать или закрывать щелчком мыши на значке узла (рис. 17.6, проект Chap_17\TTreeView\TreeViewDemo.dpr).

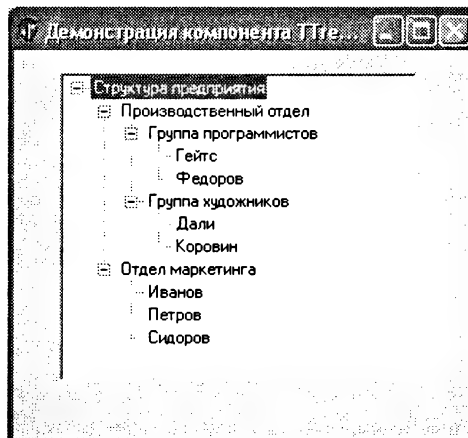


Рис. 17.6. Пример использования компонента TTreeView

Для наполнения списка на этапе разработки программы нужно щелкнуть на компоненте правой кнопкой мыши и выбрать в контекстном меню команду *Items Editor*, либо щелкнуть на нем дважды, либо, наконец, щелкнуть на кнопке с многоточием в строке свойства *Items* — во всех случаях на экране появится окно редактора компонента (рис. 17.7).

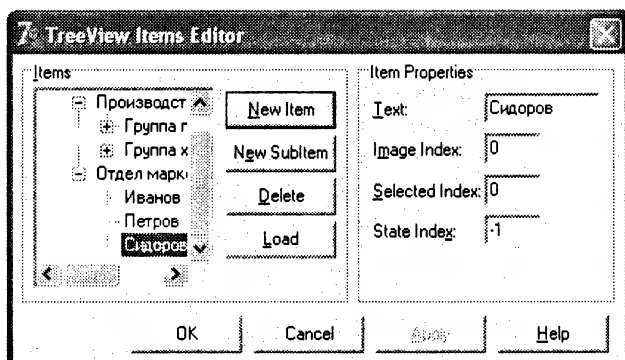


Рис. 17.7. Редактор компонента

Чтобы начать заполнение дерева, щелкните на кнопке *New Item* и введите связанный с узлом текст в поле *Text*. В поле *Image Index* раздела *Item Properties* вводятся индекс связанного с узлом значка, в поле *Selected Index* — индекс значка для выбранного узла. Для ввода дочернего узла любого уровня сначала нужно выделить (щелчком) в поле *Items* узел, который должен стать родительским, и лишь затем щелкнуть на кнопке *New SubItem*.

Для заполнения дерева в режиме прогона программы широко используется центральное свойство компонента — *Items* типа *TTreeNode*, открывающее индексированный доступ ко всем узлам списка. Каждый узел описывается классом *TTreeNode*, имеющим собственные методы и свойства. В частности, его свойство

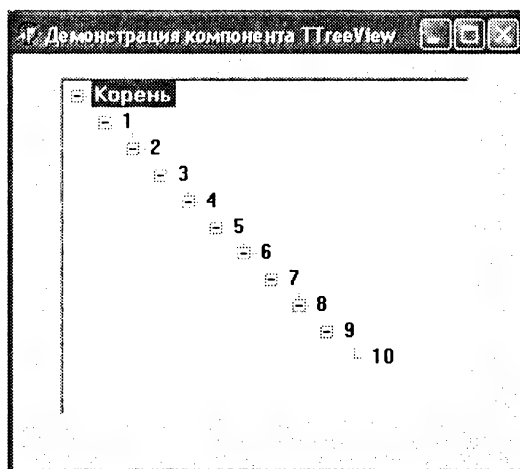


Рис. 17.8. Создание дочерних узлов на этапе прогона программы

Item содержит список всех дочерних узлов данного узла; с помощью многочисленных методов свойства TTreeView.Items к этому списку можно добавить новый дочерний узел, а с помощью метода TTreeNode.MoveTo — переместить узел в любую позицию иерархического дерева.

Процесс создания дочерних узлов на этапе прогона программы иллюстрирует рис. 17.8.

Показанную на рисунке конструкцию создал следующий обработчик события OnCreate формы (проект Chap_17\TTreeView\DynamicTree.dpr):

```
procedure TForm1.FormCreate(Sender: TObject);
var
    k: Integer;
begin
    with TTreeView1 do
        begin
            // Добавляем корневой узел
            Items.Add(NIL, 'Корень');
            // Добавляем 10 дочерних узлов
            for k := 1 to 10 do
                Items.AddChild(Items[k-1], IntToStr(k));
            // Раскрываем все узлы
            FullExpand
        end
    end;
```

Свойства компонента TTreeView представлены в табл. 17.18.

Таблица 17.18. Свойства компонента TTreeView

Свойство	Описание
TBorderStyle = bsNone..bsSingle; property BorderStyle: TBorderStyle;	Определяет стиль рамки, охватывающей компонент: bsNone — нет рамки; bsSingle — рамка толщиной в 1 пиксел
property DropTarget: TTreeNode;	Определяет узел, который может служить приемником для операций перетаскивания (Drag&Drop)
property HideSelection: Boolean;	Указывает, будет ли убираться выделение узлов, когда компонент теряет фокус ввода
property Images: TImageList;	Содержит набор изображений, которые будут использоваться при прорисовке узлов
property Indent: Integer;	Определяет отступ в пикселах от левого угла узла для всех его дочерних узлов
property Items: TTreeNodeList;	Открывает доступ к любому узлу по его индексу. Индексация начинается с нуля и соответствует просмотру всех узлов полностью развернутого дерева
property ReadOnly: Boolean;	Запрещает/разрешает редактирование надписей в узлах

— продолжение ➤

Таблица 17.18 (продолжение)

Свойство	Описание
property Selected: TTreeNode;	Содержит список всех выбранных узлов или NIL , если таких нет
property ShowButtons: Boolean;	Разрешает/запрещает показ стандартных кнопок раскрытия дочерних узлов. По умолчанию содержит значение True. Если содержит значение False, узел разворачивается двойным щелчком мыши
property ShowLines: Boolean;	Разрешает/запрещает показ линий
property ShowRoot: Boolean;	Разрешает/запрещает показ линий, идущих от самого верхнего уровня иерархии. Игнорируется, если ShowLines=False
TSortType = (stNone, stData, stText, stBoth); property SortType: TSortType;	Определяет способ сортировки узлов: stNone – нет сортировки; stData – сортировка по данным; stText – сортировка по тексту надписей; stBoth – сортировка по тексту и по данным (см. также описание события OnCompare ниже)

Методы компонента TTreeView перечислены в табл. 17.19.

Таблица 17.19. Методы компонента TTreeView

Метод	Описание
function AlphaSort: Boolean;	Сортирует узлы по тексту и возвращает значение True, если сортировка прошла успешно
TTVCompare = function (lParam1, lParam2, lParamSort: LongInt): Integer stdcall ; tfuction CustomSort (SortProc: TTVCompare; Data: LongInt): Boolean;	Определяет нестандартную сортировку с помощью функции SortProc. Эта функция должна рассматривать параметры lParam1 и lParam2 как объекты TTreeNode и возвращает отрицательное число, если lParam1 < lParam2; ноль, если lParam1 = lParam2; положительное число, если lParam1 > lParam2
procedure FullCollapse;	Прячет все узлы, кроме узлов самого верхнего уровня иерархии
Procedure FullExpand;	Показывает все узлы дерева иерархии
function GetNodeAt (X, Y: Integer): TTreeNode;	Возвращает узел, располагающийся в указанной точке, или NIL , если точка не принадлежит ни одному узлу
function IsEditing: Boolean;	Возвращает значение True, если пользователь редактирует какой-либо узел
procedure LoadFromFile (const FileName: String);	Загружает иерархическое дерево из файла
procedure SaveToFile (const FileName: String);	Сохраняет иерархическое дерево в файле
procedure SaveToStream (Stream: TStream);	Сохраняет иерархическое дерево в потоке данных

События компонента TTreeView перечислены в табл. 17.20.

Таблица 17.20. События компонента TTreeView

Событие	Описание
TTVChangedEvent = procedure (Sender: TObject; Node: TTreeNode) of object ; property OnChange: TTVChangedEvent;	Возникает при смене состояния выбора у одного из узлов. Node — узел, который изменил состояние
TTVChangingEvent = procedure (Sender: TObject; Node: TTreeNode; var AllowChange: Boolean) of object ; property OnChanging: TTVChangingEvent;	Возникает перед сменой состояния выбора у одного из узлов. Node — новый выбираемый узел. Обработчик в параметре AllowChange разрешает или запрещает возможность выделения (выбора) узла
TTVExpandedEvent = procedure (Sender: TObject; Node: TTreeNode) of object ; property OnCollapsed: TTVExpandedEvent;	Возникает при свертывании ветви дочерних узлов узла Node
TTVCollapsingEvent = procedure (Sender: TObject; Node: TTreeNode; var AllowCollapse: Boolean) of object ; property OnCollapsing: TTVCollapsingEvent;	Возникает перед свертыванием ветви дочерних узлов узла Node. В параметре AllowCollapse обработчик разрешает или запрещает свертывание
TTVCompareEvent = procedure (Sender: TObject; Node1, Node2: TTreeNode; Data: Integer; var Compare: Integer) of object ; property OnCompare: TTVCompareEvent;	Возникает при сравнении двух узлов Node1 и Node2. В параметре Compare обработчик должен вернуть отрицательное число, если Node1 < Node2, ноль, если Node1 = Node2, положительное число, если Node1 > Node2
TTVExpandedEvent = procedure (Sender: TObject; Node: TTreeNode) of object ; property OnDeletion: TTVExpandedEvent;	Возникает при удалении узла Node из иерархического дерева
TTVEditedEvent = procedure (Sender: TObject; Node: TTreeNode; var S: String) of object ; property OnEdited: TTVEditedEvent;	Возникает при завершении редактирования надписи в узле Node: S — новая надпись
TTVExpandedEvent = procedure (Sender: TObject; Node: TTreeNode) of object ; property OnExpanded: TTVExpandedEvent;	Возникает при разворачивании ветви дочерних узлов узла Node

Таблица 17.20. События компонента TTreeView

Событие	Описание
TTVExpandingEvent = procedure (Sender: TObject; Node: TTreeNode; var AllowExpansion: Boolean) of object ; property OnExpanding: TTVExpandingEvent;	Возникает перед разворачиванием ветви дочерних узлов узла Node. В параметре AllowExpansion обработчик разрешает или запрещает разворачивание
TTVExpandedEvent = procedure (Sender: TObject; Node: TTreeNode) of object ; property OnGetImageIndex: TTVExpandedEvent;	Возникает при необходимости получения индекса изображения для прорисовки узла Node в обычном состоянии
TVExpandedEvent = procedure (Sender: TObject; Node: TTreeNode) of object ; property OnGetSelectedIndex: TTVExpandedEvent;	Возникает при необходимости получения индекса изображения для прорисовки узла Node в выбранном состоянии

При программном заполнении дерева следует пользоваться свойством TTreeView.Items класса TTreeNodes. Свойства и методы этого класса представлены в табл. 17.21 и 17.22.

Таблица 17.21. Свойства класса TTreeNodes

Свойство	Описание
property Count: Integer;	Количество узлов, входящих в Items
property Item[Index: Integer]: TTreeNode; default ;	Открывает индексированный доступ к узлам
property Owner: TCustomTreeView;	Содержит ссылку на родительское дерево

Таблица 17.22. Методы класса TTreeNodes

Метод	Описание
function Add(Node: TTreeNode; const S: String): TTreeNode;	Добавляет узел в конец того дерева, в котором зарегистрирован узел Node. Если Node= NIL , добавляется корневой узел для всего компонента
function AddChild(Node: TTreeNode; const S: String): TTreeNode;	Добавляет узел в конец ветви Item дочерних узлов узла Node
function AddChildFirst(Node: TTreeNode; const S: String): TTreeNode;	Добавляет узел в начало ветви Item дочерних узлов узла Node
function AddChildObject(Node: TTreeNode; const S: String;	Добавляет узел и данные в конец ветви Item дочерних узлов узла Node. На данные

Метод	Описание
<code>Ptr: Pointer): TTreeNode;</code>	ссылается указатель <code>Ptr</code> . Связанная с данными область памяти не освобождается автоматически при удалении дерева
function <code>AddChildObjectFirst</code> (<code>Node: TTreeNode; const S: String; Ptr: Pointer</code>): <code>TTreeNode;</code>	Добавляет узел и данные в начало ветви <code>Item</code> дочерних узлов узла <code>Node</code> . На данные ссылается указатель <code>Ptr</code> . Связанная с данными область памяти не освобождается автоматически при удалении дерева
function <code>AddFirst</code> (<code>Node: TTreeNode; const S: String</code>): <code>TTreeNode;</code>	Добавляет узел в начало той ветви, в которой зарегистрирован узел <code>Node</code>
function <code>AddObject</code> (<code>Node: TTreeNode; const S: String; Ptr: Pointer</code>): <code>TTreeNode;</code>	Добавляет узел и данные в конец той же ветви, в которой зарегистрирован узел <code>Node</code> . На данные ссылается указатель <code>Ptr</code> . Связанная с данными область памяти не освобождается автоматически при удалении дерева
function <code>AddObjectFirst</code> (<code>Node: TTreeNode; const S: String; Ptr: Pointer</code>): <code>TTreeNode;</code>	Добавляет узел и данные в начало той же ветви, в которой зарегистрирован узел <code>Node</code> . На данные ссылается указатель <code>Ptr</code> . Связанная с данными область памяти не освобождается автоматически при удалении дерева
procedure <code>Assign</code> (<code>Source: TPersistent</code>);	Связывает дерево текущего компонента с деревом компонента <code>Source</code>
procedure <code>BeginUpdate</code> ;	Блокирует обновление экрана до тех пор, пока не будет выполнен метод <code>EndUpdate</code> . Используется при одновременной вставке нескольких элементов дерева для предотвращения мерцания экрана
procedure <code>Clear</code> ;	Очищает дерево от всех узлов и дочерних узлов компонента
procedure <code>Delete</code> (<code>Node: TTreeNode</code>);	Удаляет узел <code>Node</code>
procedure <code>EndUpdate</code> ;	Отменяет действие метода <code>BeginUpdate</code>
function <code>GetFirstNode</code> : <code>TTreeNode;</code>	Возвращает самый первый узел в дереве <code>Items[0]</code>
function <code>GetNode</code> (<code>ItemId: HTreeItem</code>): <code>TTreeNode;</code>	Возвращает узел по его идентификатору <code>ItemId</code>
function <code>Insert</code> (<code>Node: TTreeNode; const S: String</code>): <code>TTreeNode;</code>	Вставляет узел непосредственно перед узлом <code>Node</code>
function <code>InsertObject</code> (<code>Node: TTreeNode; const S: String; Ptr: Pointer</code>): <code>TTreeNode;</code>	Вставляет узел и данные непосредственно перед узлом <code>Node</code>

Как уже отмечалось, каждый узел класса `TTreeNode` имеет свой набор свойств (табл. 17.23) и методов (табл. 17.24).

Таблица 17.23. Свойства `TTreeNode`

Свойство	Описание
property <code>AbsoluteIndex: Integer;</code>	Возвращает абсолютный индекс узла (с учетом всех дочерних узлов)
property <code>Count: Integer;</code>	Содержит количество дочерних узлов в ветви <code>Item</code>
property <code>Cut: Boolean;</code>	Вырезает узел и помещает его в буфер обмена
property <code>Data: Pointer;</code>	Указывает на связанные с узлом данные
property <code>Deleting: Boolean;</code>	Содержит значение <code>True</code> , если для узла вызван метод <code>Destroy</code>
property <code>DropTarget: Boolean;</code>	Содержит значение <code>True</code> , если узел может служить приемником операции перетаскивания (<code>Drag&Drop</code>)
property <code>Expanded: Boolean;</code>	Содержит значение <code>True</code> , если узел развернут
property <code>Focused: Boolean;</code>	Содержит значение <code>True</code> , если узел свернут
property <code>HasChildren: Boolean;</code>	Содержит значение <code>True</code> , если узел имеет дочерние узлы
property <code>ImageIndex: TImageIndex;</code>	Содержит индекс связанного с узлом значка
property <code>Index: Longint;</code>	Содержит индекс узла в списке дочерних узлов его родительского узла
property <code>IsVisible: Boolean;</code>	Содержит значение <code>True</code> , если узел виден
property <code>Item[Index: Integer]: TTreeNode;</code>	Открывает индексированный доступ ко всем дочерним узлам
property <code>ItemId: HTreeItem;</code>	Содержит уникальный <code>Windows</code> -дескриптор узла
property <code>Level: Integer;</code>	Содержит иерархический уровень узла
property <code>OverlayIndex: Integer;</code>	Содержит индекс оверлейного значка. Оверлейный значок вычерчивается поверх основного, чтобы, например, указать, что узел стал недоступен
property <code>Owner: TTreeNode;</code>	Содержит ссылку на владельца данного узла
property <code>Parent: TTreeNode;</code>	Содержит ссылку на родительский узел
property <code>Selected: Boolean;</code>	Содержит значение <code>True</code> , если узел выделен цветом
property <code>SelectedIndex: Integer;</code>	Содержит номер значка для выделенного узла

Свойство	Описание
property Text: string;	Содержит текст узла
property TreeView: TCustomTreeView;	Содержит ссылку на компонент TTreeView, к которому принадлежит узел

Таблица 17.24. Методы класса TTreeNode

Метод	Описание
function AlphaSort: Boolean;	Сортирует узлы по алфавиту свойства Text и возвращает значение True в случае успеха
procedure Assign(Source: TPersistent); override ;	Связывает список дочерних узлов с источником Source
procedure Collapse(Recurse: Boolean);	Закрывает все узлы (Recurse=True) или только развернутые (Recurse=False)
type TTVCompare = function (lParam1, lParam2, lParamSort: Longint): Integer stdcall ; function CustomSort(SortProc: TTVCompare; Data: Longint): Boolean;	Реализует нестандартную сортировку узлов
procedure Delete;	Удаляет текущий узел
procedure DeleteChildren;	Удаляет дочерние узлы
function DisplayRect(TextOnly: Boolean): TRect;	Возвращает очерчивающий прямоугольник узла. Если TextOnly=True, возвращает очерчивающий прямоугольник текста
function EditText: Boolean;	Переводит текст узла в режим редактирования
procedure EndEdit(Cancel: Boolean);	Заканчивает редактирование текста и сохраняет его изменения, если Cancel=False
procedure Expand(Recurse: Boolean);	Развертывает узел (и его дочерние узлы, если Recurse=True)
function GetFirstChild: TTreeNode;	Возвращает ссылку на первый дочерний узел или NIL , если нет дочерних узлов
function GetLastChild: TTreeNode;	Возвращает ссылку на последний дочерний узел или NIL , если нет дочерних узлов
function GetNext: TTreeNode;	Возвращает ссылку на очередной дочерний узел
function GetNextChild(Value: TTreeNode): TTreeNode;	Возвращает ссылку на дочерний узел, следующий за узлом Value (или NIL , если такого нет)
function GetNextSibling: TTreeNode;	Возвращает ссылку на очередной узел в том же дереве

— продолжение ➤

Таблица 17.24 (продолжение)

Метод	Описание
function GetNextVisible: TTreeNode;	Возвращает ссылку на очередной видимый узел (для которого развернуты все дочерние узлы)
function GetPrev: TTreeNode;	Возвращает ссылку на предыдущий узел в том же дереве независимо от его видимости
function GetPrevChild(Value: TTreeNode): TTreeNode;	Возвращает ссылку на предыдущий по отношению к узлу Value дочерний узел
function GetPrevSibling: TTreeNode;	Возвращает ссылку на предыдущий узел того же уровня
function GetPrevVisible: TTreeNode;	Возвращает ссылку на видимый узел того же уровня
function HasAsParent(Value: TTreeNode): Boolean;	Возвращает значение True, если Value — родительский узел
function IndexOf(Value: TTreeNode): Integer;	Возвращает идентификатор узла Value
procedure MakeVisible;	Если родительский узел видимый, делает видимыми все дочерние узлы
type TNodeAttachMode = (naAdd, naAddFirst, naAddChild, naAddChildFirst, naInsert); procedure MoveTo(Destination: TTreeNode; Mode: TNodeAttachMode);	Перемещает текущий узел в позицию относительно узла Destination в зависимости от параметра Mode: naAdd — добавляет в конец списка узлов того же уровня; naAddFirst — делает первым в ветви узлов того же уровня; naAddChild — добавляет в конец списка дочерних узлов; naAddChildFirst — делает первым в списке дочерних узлов; naInsert — вставляет непосредственно перед узлом

TListView — иерархический список

Компонент TListView предназначен для отображения и выбора нескольких элементов. Каждый элемент может содержать значок и текст и подобно компоненту TTreeView иметь ветвь связанных с ним вложенных (дочерних) элементов. В отличие от компонентов TTreeView в TListView допускается не более одного уровня вложенности элементов. Компонент TListView показывает свои элементы в одной или нескольких колонках, с крупными или мелкими значками (рис. 17.9, слева). Рядом с элементами могут присутствовать флажки (рис. 17.9, справа), упрощающие множественный выбор элементов.

Компонент может наполняться как на этапе конструирования, так и на этапе прогона программы.

Выполните следующую последовательность шагов, которая продемонстрирует вам основные особенности использования компонента (проект Cbp_17\TListView\ListViewDemo.dpr).

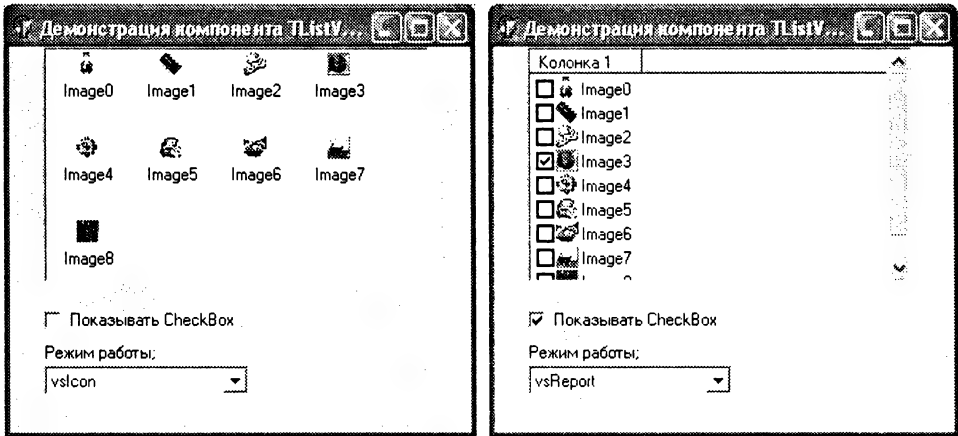


Рис. 17.9. Пример использования компонента TListView в режиме vsIcon (слева) и в режиме vsReport с показом флажков (справа)

1. На пустую форму поместите компоненты TListView, TImageList, TComboBox и TCheckBox.
2. Компонент ImageList1 будет служить контейнером для нескольких значков, отображаемых компонентом ListView. Для наполнения контейнера дважды щелкните на нем мышью или щелкните на нем правой кнопкой мыши и выберите в контекстном меню команду ImageList Editor.
3. В появившемся окне редактора изображений щелкните на кнопке Add и выберите файл Images\Icons\Chemical.
4. Повторите предыдущий шаг для выбора нескольких значков. Обратите внимание, что каждому помещаемому в компонент ImageList1 значку редактор присваивает уникальный индекс.
5. Напишите обработчики событий, представленные в листинге 17.2.

Листинг 17.2. Обработчики событий списка

```

procedure TForm1.FormCreate(Sender: TObject);
var
    k: Integer;
    ListItem: TListItem;
begin
    with ListView1 do
        begin
            // Указываем источник изображений:
            SmallImages := ImageList1;
            LargeImages := ImageList1;
            // Наполняем ListView1 значками и текстом:
            for k := 0 to ImageList1.Count - 1 do
                begin
                    // Добавляем элемент и получаем ссылку на него

```



```

    ListItem := Items.Add;
    // Указываем индекс связанного изображения
    ListItem.ImageIndex := k;
    // Вставляем надпись ImageN
    ListItem.Caption := 'Image' + IntToStr(k);
end;
// Наполняем список ComboBox1 доступными стилями
// ListView
ComboBox1.Items.AddObject('vsIcon', TObject(vsIcon));
ComboBox1.Items.AddObject('vsList', TObject(vsList));
ComboBox1.Items.AddObject('vsReport',
                           TObject(vsReport));
ComboBox1.Items.AddObject('vsSmallIcon',
                           TObject(vsSmallIcon));
// Показываем первый стиль в Combo Box
ComboBox1.ItemIndex := 0;
// Создаем колонку для режима ysReport:
Columns.Add;
Columns[0].Caption := 'Колонка 1';
Columns[0].Width := 80;
end;
end;
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
    with ComboBox1 do
        ListView1.ViewStyle := TViewStyle(Items.Objects
        [ItemIndex]);
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
    ListView1.CheckBoxes := CheckBox1.Checked
end;

```

Некоторые важные свойства компонента TListView представлены в табл. 17.25.

Таблица 17.25. Свойства компонента TListView

Свойство	Описание
property AllocBy: Integer;	Используется перед вставкой большого количества элементов: для экономии времени и памяти перед добавлением элементов установите в это свойство их количество
property CheckBoxes: Boolean;	Разрешает/запрещает показ флажков рядом с элементами
property Column[Index: Integer]: TListColumn;	Открывает доступ к колонкам элементов по их индексам

Свойство	Описание
property ColumnClick: Boolean;	Разрешает/запрещает генерацию события OnColumnClick
property Columns: TListColumns;	Содержит объекты-колонок. Используйте это свойство для удаления или добавления колонок, а также для изменения их свойств. Колонки видны только в режиме ViewStyle=vsReport. И наоборот: элементы в этом режиме видны, только если определена хотя бы одна колонка
property GridLines: Boolean;	Разрешает/запрещает показ линий в режиме ViewStyle=vsReport
property HideSelection: Boolean;	Запрещает/разрешает сохранять элементы выделенными при потере компонентом фокуса ввода
property HotTrack: Boolean;	Разрешает/запрещает подсветку элементов, на которых проходит указатель мыши
property IconOptions: TIconOptions;	С помощью объекта класса TIconOptions задаются дополнительные условия отображения (см. ниже)
property Items: TListItems;	Содержит список всех элементов
property LargeImages: TImageList;	Указывает источник крупных значков
property MultiSelect: Boolean;	Разрешает/запрещает множественный выбор
property ReadOnly: Boolean;	Запрещает/разрешает редактирование надписей
property ShowColumnHeaders: Boolean;	Разрешает/запрещает показ заголовков колонок в режиме ViewStyle=vsReport
property SmallImages: TImageList;	Определяет источник мелких значков
TSortType = (stNone, stData, stText, stBoth); property SortType: TSortType;	Определяет способ сортировки элементов
property StateImages: TImageList;	Определяет источник значков для выбранных элементов
TViewStyle = (vsIcon, vsSmallIcon, vsList, vsReport); property ViewStyle: TViewStyle;	Определяет стиль показа элементов: vsIcon — крупные значки; vsSmallIcon — мелкие значки; vsList — список значков; vsReport — таблица значков

ВНИМАНИЕ Чтобы компонент работал в режиме vsReport, необходимо создать хотя бы одну колонку. На этапе конструирования программы колонки создаются и изменяются с помощью редактора колонок, окно которого открывается после щелчка на компоненте правой кнопкой мыши и выборе в контекстном меню команды Column Editor. Пример программного создания колонки был показан в листинге 17.2.

Класс TIconOption определяет дополнительные условия отображения узлов; свойства этого класса представлены в табл. 17.26.

Таблица 17.26. Свойства класса TiconOption

Свойство	Описание
type TIconArrangement = (iaTop, iaLeft); property Arrangement: TIconArrangement;	Определяет способ расположения изображений: iaTop — слева направо (вверху колонки); iaLeft — сверху вниз слева от ряда колонок
property AutoArrange: Boolean;	Если содержит значение True, элементы будут автоматически располагаться правильными рядами при изменении их количества
property WrapText: Boolean;	Если содержит значение True, текст элемента может переноситься по словам на несколько строк, если он выходит за границы значка

При программном заполнении компонента центральную роль играет свойство Items класса TListItems (табл. 17.27).

Таблица 17.27. Свойства класса TListItems

Свойство	Описание
property Count: Integer;	Содержит количество элементов в списке Item
property Item[Index: Integer]: TListItem;	Открывает индексный доступ к элементам списка

Методы класса TListItems представлены в табл. 17.28.

Таблица 17.28. Методы класса TListItems

Метод	Описание
function Add: TListItem;	Добавляет очередной элемент к списку
procedure Assign(Source: TPersistent); override ;	Связывает список компонента со списком компонента Source
procedure BeginUpdate;	Блокирует обновление экрана до тех пор, пока не будет выполнен метод EndUpdate. Используется при одновременной вставке нескольких элементов списка для предотвращения мерцания экрана

Метод	Описание
procedure Clear;	Очищает список
procedure Delete (Index: Integer);	Удаляет элемент списка с индексом Index
procedure EndUpdate;	Отменяет действие BeginUpdate
function IndexOf (Value: TListItem): Integer;	Возвращает индекс узла Value
function Insert (Index: Integer): TListItem;	Вставляет новый элемент после элемента, заданного индексом Index
procedure SetCount (Value: Integer);	Устанавливает новое количество элементов списка

THeaderControl — управляющий заголовок

Компонент `THeaderControl` представляет собой многоколончатый заголовок с регулируемыми размерами колонок (секций). Каждая колонка (секция) заголовка может содержать текст и/или графику. Компонент способен обрабатывать событие `OnResize`, которое возникает при каждом изменении размеров любой секции. В ходе обработки этого события программа обычно соответствующим образом изменяет линейные размеры столбцов таблицы или подобной структуры, с которой связан компонент.

В программе, окно которой показано на рис. 17.10 (проект `Chap_17\HeaderControl\HeadCont.dpr`), компонент `THeaderControl` используется для управления положением и линейными размерами трех других компонентов.

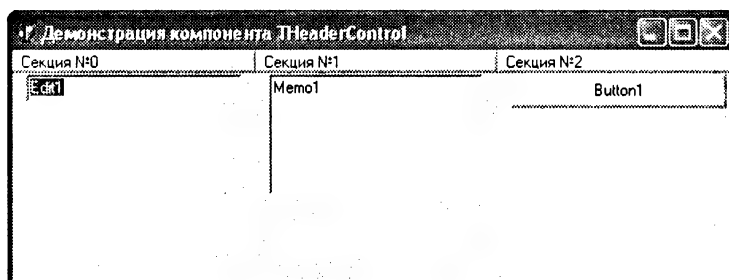


Рис. 17.10. Пример использования компонента `THeaderControl` для управления положением и размерами других компонентов

Для реализации программы выполните следующую последовательность шагов.

1. Поместите на пустую форму компоненты `THeaderControl`, `TEdit`, `TMemo` и `TButton`. По умолчанию должно выполняться условие `HeaderControl1.Align = alTop` — убедитесь в этом и установите свойство `Align`, если это не так. Положение и размеры других компонентов не имеют значения.
2. Создайте обработчики событий `OnCreate` для формы `Form1`, а также `OnResize` и `OnSectionResize` для компонента `HeaderControl1` (листинг 17.3).

Листинг 17.3. Обработчики событий для формы и для заголовка

```

const
    Delta = 10; // Зазор между границами заголовка
               // и компонентами
procedure TForm1.FormCreate(Sender: TObject);
var
    HSection: THeaderSection;
    k: Integer;
begin
    // Создаем три секции заголовка:
    with HeaderControl1 do for k := 0 to 2 do
        begin
            HSection := Sections.Add;
            HSection.Text := 'Секция №' + IntToStr(k);
            HSection.Width := Form1.Width div 3;
            HSection.MinWidth := 3 * Delta;
        end;
    end;

procedure TForm1.HeaderControl1Resize(Sender: TObject);
    // Устанавливает положение и размеры компонентов
begin
    with HeaderControl1 do
        begin
            Edit1.Left := Delta;
            Edit1.Top := HeaderControl1.Height + 1;
            Edit1.Width := Sections.Items[0].Width - 2 * Delta;

            Mem1.Left := Sections.Items[1].Left + Delta;
            Mem1.Top := HeaderControl1.Height + 1;
            Mem1.Width := Sections.Items[1].Width - 2 * Delta;

            Button1.Left := Sections.Items[2].Left + Delta;
            Button1.Top := HeaderControl1.Height + 1;
            Button1.Width := Sections.Items[2].Width - 2 * Delta;
        end
    end;

procedure TForm1.HeaderControl1SectionResize(
    HeaderControl: THeaderControl; Section: THeaderSection);
begin
    HeaderControl1Resize(Self)
end;

```

Не забудьте определить в области видимости обработчиков FormCreate и HeaderControl1Resize глобальную константу Delta.

Для заголовка создается объект класса `THeaderSections`, определяющий секции заголовка. Следующее свойство этого объекта открывает доступ к индексированным объектам-секциям класса `THeaderSection`:

property `Items[Index: Integer]: THeaderSection;`

Свойства класса `THeaderSection` перечислены в табл. 17.29.

Таблица 17.29. Свойства класса `THeaderSection`

Свойство	Описание
property <code>Alignment: TAlignment;</code>	Определяет выравнивание текста: <code>taLeftJustify</code> — выравнивание влево; <code>taCenter</code> — текст центрирован по горизонтали; <code>taRightJustify</code> — выравнивание вправо
property <code>AllowClick: Boolean;</code>	Разрешает/запрещает генерацию события <code>OnSectionClick</code> при щелчке на секции
property <code>Left: Integer;</code>	Определяет положение левой границы секции относительно границ компонента (в пикселах)
property <code>MaxWidth: Integer;</code>	Определяет максимальную ширину секции в пикселах
property <code>MinWidth: Integer;</code>	Определяет минимальную ширину секции в пикселах
property <code>Right: Integer;</code>	Определяет положение правой границы секции относительно границ компонента (в пикселах)
<code>THeaderSectionStyle = (hsText, hsOwnerDraw);</code> property <code>Style: THeaderSectionStyle;</code>	Определяет способ формирования секции: <code>hsText</code> — секция содержит только текст и изображается автоматически; <code>hsOwnerDraw</code> — секция прорисовывается программой
Property <code>Width: Integer;</code>	Содержит текущую ширину секции в пикселах

Следующий метод добавляет очередную секцию заголовка:

function `THeaderSections.Add: THeaderSection;`

Некоторые свойства класса `THeaderSection` представлены в табл. 17.30.

Таблица 17.30. Свойства класса `THeaderSection`

Свойство	Описание
property <code>AllowClick: Boolean;</code>	Если содержит значение <code>True</code> , разрешает щелкнуть на секции заголовка, как на кнопке. В этом случае управление передается обработчику события <code>OnSectionClick</code>
property <code>DisplayName: String;</code>	Имя заголовка секции
property <code>ImageIndex: TImageIndex;</code>	Определяет индекс изображения из хранилища <code>Images</code> , которое будет появляться следом за текстом заголовка

— продолжение ↗

Таблица 17.30 (продолжение)

Свойство	Описание
property MaxWidth: Integer; property MinWidth: Integer;	Определяет максимальную и минимальную ширину секции
type THeaderSectionStyle = (hsText, hsOwnerDraw); property Style: THeaderSectionStyle;	Определяет тип секции: hsText — секция содержит только текст; hsOwnerDraw — секция нуждается в программной прорисовке
property Text: String;	Содержит текст секции

Свойства компонента THeaderControl представлены в табл. 17.31.

Таблица 17.31. Свойства компонента THeaderControl

Свойство	Описание
property Canvas: TCanvas;	С помощью этого свойства можно изображать графику в секциях заголовка
property HotTrack: Boolean;	Разрешает/запрещает цветовое выделение секции, на которой располагается указатель мыши
property Sections: THeaderSections;	Содержит секции заголовка

События компонента THeaderControl перечислены в табл. 17.32.

Таблица 17.32. События компонента THeaderControl

Событие	Описание
TDrawSectionEvent = procedure (HeaderControl: THeaderControl; Section: THeaderSection; const Rect: TRect; Pressed: Boolean) of object ; property OnDrawSection: TDrawSectionEvent;	Возникает при необходимости прорисовать секцию, для которой установлен стиль hsOwnerDraw: Section — объект-секция; Rect — прямоугольник прорисовки; Pressed — признак нажатой кнопки мыши
property OnResize: TNotifyEvent;	Возникает при изменении размеров компонента
TSectionNotifyEvent = procedure (HeaderControl: THeaderControl; Section: THeaderSection) of object ; property OnSectionClick: TSectionNotifyEvent;	Возникает при щелчке мышью на секции Section
TSectionNotifyEvent = procedure (HeaderControl: THeaderControl; Section: THeaderSection) of object ; property OnSectionResize: TSectionNotifyEvent;	Возникает при изменении размеров секции Section

Событие	Описание
TSectionTrackState = (tsTrackBegin, tsTrackMove, tsTrackEnd); TSectionTrackEvent = procedure (HeaderControl: THeaderControl; Section: THeaderSection; Width: Integer; State: TSectionTrackState) of object ; property OnSectionTrack: TSectionTrackEvent;	Возникает при изменении размеров секции и позволяет обработать три возможных состояния: tsTrackBegin — начало перемещения границы; tsTrackMove — идет перемещение границы; tsTrackEnd — окончание перемещения границы

TStatusBar — строка состояния

Компонент TStatusBar предназначен для создания строки состояния, которая обычно располагается в нижней части основной формы. Компонент может иметь несколько панелей (секций), а также кнопку изменения размеров окна, в которое он помещен (рис. 17.11).

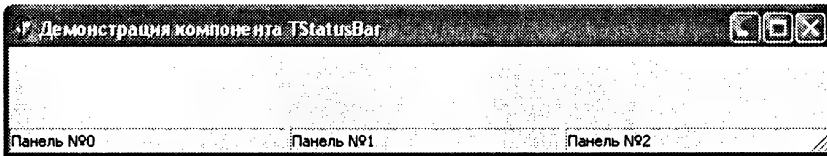


Рис. 17.11. Пример компонента с тремя панелями и кнопкой изменения размеров окна

Показанный на рисунке компонент TStatusBar с тремя панелями и кнопкой изменения размеров окна создан следующим обработчиком события OnCreate формы Form1 (предварительно на пустую форму поместите компонент TSatus-Bar):

```
procedure TForm1.FormCreate(Sender: TObject);
var
    Panel: TStatusBarPanel;
    k: Integer;
begin
    with StatusBar1 do for k := 0 to 2 do
        begin
            Panel := Panels.Add;
            Panel.Text := Format('Панель №%d', [k]);
            Panel.Width := Form1.Width div 3
        end;
    end;
```

С компонентом связывается объект класса TStatusBarPanels, который определяет панели компонента. Каждая панель относится к классу TStatusBarPanel и имеет свойства, перечисленные в табл. 17.33.

Таблица 17.33. Свойства класса TStatusPanel

Свойство	Описание
property Alignment: TAlignment;	Определяет выравнивание текста относительно границ панели; taLeftJustify — выравнивается влево; taCenter — центрируется по горизонтали; taRightJustify — выравнивается вправо
property Bevel: TStatusPanelBevel;	Определяет стиль рамки панели; pbNone — нет рамки; pbLowered — вдавленная рамка; pbRaised — выпуклая рамка
property Style: TStatusPanelStyle;	Определяет способ формирования изображения панели: psText — панель содержит только текст и прорисовывается автоматически; psOwnerDraw — панель прорисовывается программой
property Text: String;	Определяет текст надписи в панели
property Width: Integer;	Определяет ширину панели в пикселах

Свойства компонента TStatusBar перечислены в табл. 17.34.

Таблица 17.34. Свойства компонента TStatusBar

Свойство	Описание
property Canvas: TCanvas;	Канва для прорисовки панелей
property Panels: TStatusPanels;	Содержит объекты-панели. Свойство TStatusPanels.Items[Index: Integer]: TstatusPanel открывает доступ к панели по ее индексу
property SimplePanel: Boolean;	Запрещает/разрешает создание нескольких панелей. Если содержит значение True, компонент имеет единственную панель
property SimpleText: String;	Содержит текст панели для SimplePanel=True
property SizeGrip: Boolean;	Разрешает/запрещает вставку кнопки изменения размеров окна. Игнорируется, если Align<>alBottom

События компонента TStatusBar представлены в табл. 17.35.

Таблица 17.35. События компонента TStatusBar

Событие	Описание
TDrawPanelEvent = procedure (StatusBar: TStatusBar; Panel: TStatusPanel; const Rect: TRect) of object ; property OnDrawPanel: TDrawPanelEvent;	Возникает при необходимости прорисовки панели Panel, если ее свойство Style определено как psOwnerDraw: Rect — прямоугольник прорисовки
Property OnResize: TNotifyEvent;	Возникает при изменении размеров компонента

TToolBar и TToolButton — панель инструментов и кнопки для нее

Компонент `TToolBar` представляет собой специальный контейнер для создания панелей инструментов. Главная отличительная черта компонента `TToolBar` — его способность гибкого управления дочерними элементами, которые он может группировать, выравнивать по размерам, располагать в несколько рядов. Компонент может манипулировать любыми вставленными в него дочерними элементами, но все его возможности в полной мере проявляются только со специально для него разработанным компонентом `TToolButton` (кнопка панели инструментов). Этот компонент похож на кнопку `TSpeedButton`, но не ищите его в палитре компонентов, — его там нет. Поскольку он разработан специально для компонента `TToolBar`, вставить его в панель инструментов можно только после щелчка правой кнопкой мыши на компоненте `TToolBar` и выборе в контекстном меню команды `New Button` (Новая кнопка) или `New Separator` (Новый разделитель) — разделители предназначены для функционального выделения на панели инструментов групп элементов и представляют собой разновидности кнопок `TToolButton`. Компонент `TToolButton` не имеет свойства, предназначенного для хранения изображения, однако компонент `TToolBar` позволяет извлечь нужное изображение из контейнера `TImageList` и поместить это изображение на кнопку.

Методику использования компонентов `TToolBar` и `TToolButton` рассмотрим на примере программы, окно которой показано на рис. 17.12.



Рис. 17.12. Пример панелей `TToolBar` с кнопками `TToolButton`

1. Поместите на пустую форму компонент `TImageList`, щелкните на нем правой кнопкой мыши и выберите команду `ImageList Editor`.
2. Для заполнения контейнера `ImageList1` подойдут любые небольшие по размеру изображения. К сожалению, растры из каталога `Images\Buttons` рассчитаны на использование в кнопках `TSpeedButton` и содержат по два изображения каждый — обычное и для недоступной кнопки. Кнопки `TToolButton` также могут иметь два или три изображения, но эти изображения необходимо размещать в разных контейнерах `TImageList`, поэтому использование заготовок из каталога `Images\Buttons` в кнопках `TToolButton` приведет к появлению на них двух изображений одновременно¹. Чтобы этого не произошло, загрузите значок из каталога `Images\Icons`, щелкнув на кнопке `Add` в окне редактора и выбрав в этом каталоге файл `Chemical`.

¹ Точнее, при вставке двойного изображения редактор компонента спросит у вас, надо ли его разделить на два отдельных изображения.

3. Повторите предыдущий шаг для файлов Chip, Construc, Earth, Finance, Handcek, Shipping, Skyline и Technlgy, после чего закройте редактор компонента ImageList.
4. Поместите на форму компонент TToolBar, в окне инспектора объектов разыщите свойство Images, раскройте список в правой колонке свойства Images и выберите пункт ImageList1. Таким образом мы указали компоненту на источник изображений и можем теперь вставлять в него кнопки.
5. Щелкните на компоненте ToolBar1 правой кнопкой мыши и выберите команду New Button.
6. Вставьте еще две кнопки, после чего вставьте разделитель командой New Separator.
7. Разместите на компоненте ToolBar1 остальные кнопки и разделители (см. рис. 17.12).
8. Если хотите, чтобы кнопки имели модный «плоский» вид, установите в свойство Flat компонента ToolBar1 значение True.

Свойство Buttons компонента TToolBar позволяет обращаться к каждому дочернему компоненту как к объекту класса TToolButton, свойства которого представлены в табл. 17.36.

Таблица 17.36. Свойства класса TToolButton

Свойство	Описание
property AllowAllUp: Boolean;	Если содержит значение True, синхронизирует свое состояние с состоянием других кнопок в той же группе: в любой момент может быть нажата только одна кнопка группы. Игнорируется, если Grouped=False
property Caption: String;	Содержит связанный с кнопкой текст, который будет показан, если свойство ShowCaptions компонента TToolBar имеет значение True
property Down: Boolean;	Определяет состояние кнопки: если содержит значение True, кнопка утоплена
property DropdownMenu: TPopupMenu;	Связывает с нажатой кнопкой контекстное меню
property Grouped: Boolean;	Разрешает/запрещает учитывать свойство AllowAllUp
property ImageIndex: Integer;	Определяет индекс связанного с кнопкой изображения
property Indeterminate: Boolean;	Запрещает/разрешает выбор кнопки
TToolButtonStyle = (tbsButton, tbsCheck, tbsDropDown, tbsSeparator, tbsDivider); property Style: TToolButtonStyle;	Определяет стиль кнопки: tbsButton — обычная кнопка; tbsCheck — фиксируемая кнопка (остается в нажатом положении, для ее освобождения нужно щелкнуть на ней еще раз); tbsDropDown — кнопка с символом раскрывающегося списка; tbsSeparator — разделитель (на месте этой кнопки будет пустое место); tbsDivider — разделитель

Свойство	Описание
	(в работающей программе на месте этой кнопки будет вертикальная черта)
property Wrap: Boolean;	Если имеет значение True, кнопка завершает текущий ряд кнопок. Игнорируется, если свойство Wrapable компонента TToolBar имеет значение True

Свойства компонента TToolBar перечислены в табл. 17.37.

Таблица 17.37. Свойства компонента TToolBar

Свойство	Описание
property AutoSize: Boolean;	Если содержит значение True, высота компонента будет автоматически согласовываться с высотой кнопок
property ButtonCount: Integer;	Содержит количество вставленных в панель инструментов дочерних компонентов (не только кнопок TToolButton)
property ButtonHeight: Integer;	Определяет высоту кнопок TToolButton
property DisabledImages: TImageList;	Определяет контейнер для изображений недоступных кнопок
property Flat: Boolean;	Если содержит значение True, панель и кнопки на ней прозрачны для фона и вокруг кнопки появляется граница, только когда на ней располагается указатель мыши
property HotImages: TImageList;	Определяет контейнер изображений для кнопок в момент, когда на кнопке располагается указатель мыши
property Images: TImageList;	Определяет контейнер для изображений кнопок в обычном состоянии
property Indent: Integer;	Определяет отступ в пикселах от левого края компонента для первого дочернего элемента
property List: Boolean;	Если имеет значение True, изображение выравнивается по левой границе кнопки, а текст — по правой, в противном случае текст выводится под изображением. Игнорируется, если ShowCaptions=False
property RowCount: Integer;	Содержит количество рядов кнопок
property ShowCaptions: Boolean;	Разрешает/запрещает показ на кнопках текста
property Wrapable: Boolean;	Запрещает/разрешает располагать кнопки в нескольких рядах

Для компонента определено событие OnResize, возникающее при изменении размеров компонента.

TCoolBar и TCoolBand — панель инструментов и полосы для нее

Компонент TCoolBar предназначен для создания настраиваемых панелей инструментов. Для каждого размещаемого на компоненте TCoolBar элемента создается объект класса TCoolBand (полоса), который может изменять свои размеры и положение в пределах границ компонента.

Центральным свойством компонента является свойство Bands — массив созданных в компоненте полос TCoolBand. Каждая полоса может иметь текст, значок и произвольный элемент управления. В отличие от компонента TToolBar или TControlBar полоса в компоненте TCoolBar всегда содержит только один интерфейсный элемент, но ничто не мешает этому элементу быть элементом-контейнером для размещения нескольких компонентов. Размещенный на полосе компонент всегда стремится занять левый верхний угол полосы, при этом остальные размеры полосы изменяются так, чтобы полностью охватить элемент.

Поясним сказанное несложным примером.

1. Поместите на пустую форму компонент TCoolBar — он тотчас же займет верхнюю часть формы, так как по умолчанию его свойство Alignment содержит значение alTop.
2. Щелкните на компоненте левой кнопкой мыши и выберите в контекстном меню команду Bands Editor (Редактор полос).
3. Окно редактора похоже на многие другие используемые в Delphi окна редакторов: щелкните на кнопке, чтобы вставить в компонент новую полосу.
4. В окне инспектора объектов можете задать текст для полосы и связанный с ней значок свойствами соответственно Text и ImageIndex (предварительно компонент в свойстве Images должен уже содержать ссылку на связанный с ним контейнер TImageList). Свойство BitMap используется для заполнения полосы чередующимся узором. Для простоты ограничьтесь текстом Полоса 1 и закройте редактор полос.
5. Поместите ниже полосы, но в тот же контейнер CoolBar1, любой элемент управления, например, текстовое поле TEdit. Подобно полосе поле растянется вдоль верхнего края контейнера и рядом появится вешка перемещения для реализации механизма перетаскивания (Drag&Dock).
6. Вновь раскройте редактор полос, выделите ранее установленную полосу и в инспекторе объектов раскройте список свойства Control — в нем окажется только что вставленный в форму компонент TEdit. Щелкните на нем — и он «перепрыгнет» на первую полосу, оставив ранее занимаемую им полосу TCoolBar пустой. Таким способом происходит связывание полосы TCoolBar с компонентом.

Освободившуюся полосу можно связать с новым элементом управления (например, ComboBox) и т. д.

Окна работающей программы показаны на рис. 17.13.

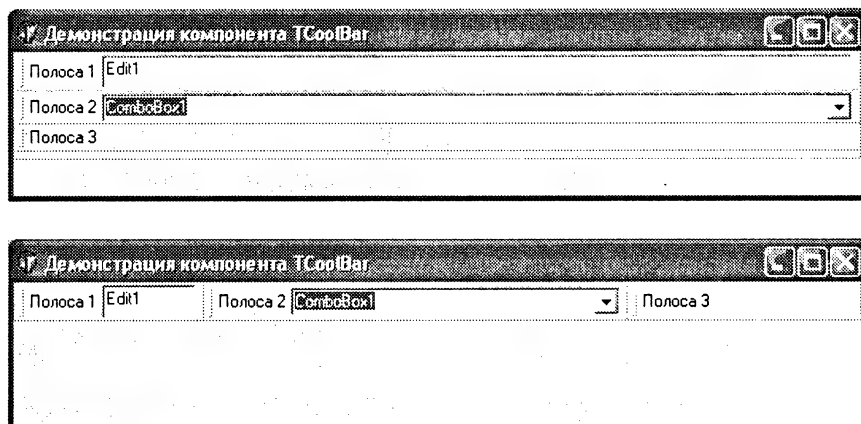


Рис. 17.13. Вид компонента CollBar при разных настройках

Свойства компонента TCoolBand представлены в табл. 17.38.

Таблица 17.38. Свойства компонента TCoolBand

Свойство	Описание
property Bitmap: TBitmap;	Определяет изображение, которое будет циклически повторяться по всему пространству полосы
property Break: Boolean;	Если содержит значение True, полоса располагается в новой строке, в противном случае — в той же строке, что и предыдущая полоса
Property Control: TWinControl;	Связанный с полосой элемент управления
property FixedBackground: Boolean;	Запрещает/разрешает периодическое повторение изображения Bitmap по всей поверхности полосы
property FixedSize: Boolean;	Запрещает/разрешает изменение размеров полосы
property HorizontalOnly: Boolean;	Предписывает показывать полосу только горизонтально
property ImageIndex: Integer;	Содержит индекс связанного с полосой изображения
property MinHeight: Integer;	Определяет минимальное значение высоты полосы при изменении ее размеров
property MinWidth: Integer;	Определяет минимальное значение ширины полосы при изменении ее размеров
property ParentBitmap: Boolean;	Разрешает/запрещает использовать значение свойства Bitmap компонента-владельца TCoolBar вместо собственного свойства Bitmap
property Text: String;	Содержит текст полосы

Свойства компонента TCoolBar перечислены в табл. 17.39.

Таблица 17.39. Свойства компонента TCoolBar

Свойство	Описание
property AutoSize: Boolean;	Если содержит значение True, высота компонента будет автоматически согласовываться с высотой полос
property Bands: TCoolBands;	Содержит список всех полос. Свойство Items этого объекта открывает доступ к полосе по ее индексу
property Bitmap: TBitmap;	Определяет изображение, которое будут использовать все полосы
property FixedOrder: Boolean;	Запрещает/разрешает перемещение полос
property FixedHeight: Boolean;	Запрещает/разрешает изменение размера полос
property Images: TImageList;	Указывает контейнер для изображений, связанных с каждой полосой
property ShowText: Boolean;	Если содержит значение True, на полосе отображается связанный с ней текст
property Vertical: Boolean;	Если содержит значение True, полосы располагаются вдоль вертикальной границы компонента

События компонента TCoolBar перечислены в табл. 17.40.

Таблица 17.40. События компонента TCoolBar

Событие	Описание
property OnChange: TNotifyEvent;	Возникает при изменении свойств Break, Index или Width у любой полосы
property OnResize: TNotifyEvent;	Возникает при изменении размеров компонента

TPageScroller — панель с кнопками прокрутки

Компонент служит контейнером для размещения длинных узких компонентов наподобие компонента TToolBar. Его отличительная черта — наличие небольших стрелок прокрутки по сторонам контейнера в случае, если ширина (высота) дочернего окна превышает соответствующий размер компонента. Чтобы увидеть компонент в действии, перейдите во вкладку Win32 палитры компонентов Delphi — сам компонент TPageScroller на этой вкладке обычно не виден, и, чтобы получить к нему доступ, следует щелкнуть на кнопке с небольшой стрелкой в правой части страницы.

По назначению компонент весьма напоминает уже рассмотренный компонент TScrollBar (см. раздел «TScrollBar — панель с полосами прокрутки» в главе 16) и отличается от него двумя обстоятельствами: компонент не содержит полос про-

крутки, а прокрутка осуществляется так, чтобы полностью показать ранее невидимый дочерний компонент.

Свойства компонента TPageScroller представлены в табл. 17.41.

Таблица 17.41. Свойства компонента TPageScroller

Свойство	Описание
property AutoScroll: Boolean;	Разрешает/запрещает автоматическую прокрутку при остановке указателя мыши на кнопке прокрутки
property ButtonSize: Integer;	Определяет размер в пикселах кнопок прокрутки: для горизонтальной ориентации — ширину, для вертикальной — высоту
property Control: TWinControl;	Содержит ссылку на прокручиваемое окно
property DragScroll: Boolean;	Указывает, будет ли компонент приемником в операциях перетаскивания (Drag&Drop). Если содержит значение True, компонент будет прокручивать дочернее окно при перемещении на нем указателя мыши с «грузом»
property Margin: Integer;	Указывает расстояние в пикселах от границы компонента до соответствующей стороны дочернего окна
type TPageScrollerOrientation = (soHorizontal, soVertical); property Orientation: TPageScrollerOrientation;	Определяет ориентацию компонента: soHorizontal — по горизонтали; soVertical — по вертикали
property Position: Integer;	Определяет текущее положение прокручиваемого окна относительно границ компонента

TComboBoxEx — комбинированный список с расширенными возможностями

Компонент TComboBoxEx впервые введен в версии 6. По сравнению с рассмотренным компонентом TComboBox (см. раздел «TComboBox — комбинированный список» в главе 15), он позволяет снабжать пункты (элементы) списка небольшими изображениями, а также регулирует отступ каждого элемента от левого края списка. Пример компонента показан на рис. 17.14 (проект Chap_17\ComboBoxEx\ComboEx.dpr).

1. Поместите на пустую форму компоненты TComboBoxEx и TImageList. В свойство TComboBoxEx.Images поместите ссылку на ImageList1 — этот компонент будет хранилищем изображений для ComboBoxEx1.
2. Наполните компонент ImageList1 несколькими изображениями, как это описано, например, в разделе «TListView — иерархический список».

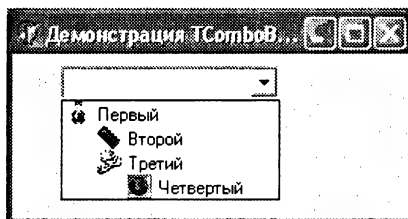


Рис. 17.14. Демонстрация компонента TComboBoxEx

Чтобы повторить пример, проделайте следующие шаги.

3. В окне инспектора объектов щелкните на кнопке с многоточием в строке сложного свойства `ComboBoxEx1.ItemsEx`, чтобы раскрыть редактор этого свойства.
4. Добавьте с помощью редактора столько строк в список, сколько изображений хранится в компоненте `ImageList1` (если, разумеется, в каждую строку вы хотите поместить собственное изображение). Для каждой строки в окне инспектора объектов задайте текст (свойство `Caption`), индекс изображения (`ImageIndex`), смещение от левого края (`Indent`) и индекс изображения для выбранной строки (`SelectImageIndex`). Отрицательное или нулевое значение `Indent` означает отсутствие отступа, любое положительное число определяет отступ в пикселах горизонтального размера изображений. Если значение свойств `SelectImageIndex` и `ImageIndex` не совпадают, выбранная строка получит не то изображение, которое было в раскрытом списке в момент выбора. Это же изображение она получит в момент повторного раскрытия списка, но при выборе другой строки вернет начальное изображение.

Следует заметить, что функциональность компонент не полностью соответствует функциональности компонента `TComboBox`, так как в нем отсутствуют некоторые возможности компонента `TComboBox` (например, программная прорисовка строк). Большая часть свойств, методов и событий компонентов `TComboBoxEx` и `TComboBox` совпадают. Специфические свойства описаны в табл. 17.42.

Таблица 17.42. Свойства компонента TComboBoxEx

Свойство	Описание
property <code>Images:</code> <code>TCustomImageList;</code>	Содержит ссылку на объект-хранилище изображений
property <code>ItemsEx:</code> <code>TComboExItems;</code>	Содержит ссылку на объект класса <code>TComboExItems</code>

Объект класса `TComboExItems` содержит всю необходимую информацию о списке в целом. Свойства этого класса представлены в табл. 17.43.

Таблица 17.43. Свойства класса TComboExItems

Свойство	Описание
property <code>CaseSensitive:</code> <code>Boolean;</code>	Определяет, будут ли учитываться регистр букв при сортировке строк
property <code>Count:</code> <code>Integer;</code>	Содержит количество строк в списке

Свойство	Описание
property ComboItems[const Index: Integer]: TComboBoxItem;	Открывает индексированный доступ к объектам класса TComboBoxItem, в которых хранится информация о каждом элементе списка
type TListItemsSortType = (stNone, stData, stText, stBoth); property SortType: TSortType;	Определяет способ сортировки элементов: stNone — нет сортировки; stData — по значениям свойства Data; stText — по значениям свойства Caption; stBoth — объединенная сортировка (по значениям обоих свойств). Свойства Data и Caption представлены ниже в описании класса TComboBoxItem

Если для списка указан тип сортировки stText, строки списка сортируются немедленно. Если указан тип stData или stBoth, сортировка осуществляется в обработчике события TComboBoxItems.OnCompare.

В объектах класса TComboBoxItem хранится вся информация о каждом элементе списка. Этот класс имеет собственные свойства (табл. 17.44).

Таблица 17.44. Свойства класса TComboBoxItem

Свойство	Описание
property Caption: String;	Текст элемента списка
property Data: Pointer; произвольный объект	Указатель на связанный с элементом
type TImageIndex = type Integer; property ImageIndex: TImageIndex;	Индекс связанного с элементом изображения в раскрытом списке
property Indent: Integer;	Смещение элемента от правого края листа ¹
property OverlayImageIndex: TImageIndex;	Определяет индекс маски, накладываемой на изображения ImageIndex и SelectedImageIndex
property SelectedImageIndex: TImageIndex;	Индекс изображения для выбранного элемента списка

У класса TComboBoxItems есть собственные методы (табл. 17.45) и события, управляющие списком в целом.

Таблица 17.45. Методы класса TComboBoxItems

Метод	Описание
function Add: TComboBoxItem;	Добавляет к списку новый элемент
function AddItem(const Caption: String; const ImageIndex, SelectedImageIndex,	Добавляет к списку новый элемент и инициализи-

продолжение ➤

¹ В документации сказано, что отступ определяется в пикселах экрана. На самом деле это не так — он приблизительно равен ширине изображений: 1 — отступ на одно изображение, 2 — на два и т. д.

Таблица 17.45 (продолжение)

Метод	Описание
OverlayImageIndex, Indent: Integer; Data: Pointer): TComboExItem;	рулет его свойства
function Insert(Index: Integer): TComboExItem;	Вставляет в указанную позицию списка новый элемент
type TListItemsCompare = function (List: TListControlItems; Index1, Index2: Integer): Integer; procedure CustomSort(Compare: TListItemsCompare);	Сортирует элементы с помощью функции Compare
procedure Sort;	Сортирует элементы с помощью обработчика события OnCompare

Помимо перечисленных в табл. 17.45, в классе определены также типичные для любой коллекции методы: Assign, Clear, Delete и т. д.

Для класса определено единственное событие:

```
type TListCompareEvent = function(List: TListControlItems;  
    tem1, Item2: TListControlItem): Integer of object;  
property OnCompare: TListCompareEvent;
```

Это событие возникает, если в качестве типа сортировки выбрано значение stData или stBoth. Обработчик должен проанализировать два элемента и вернуть -1, 0 или +1, если соответственно первый элемент меньше, равен или больше второго.

Методы компонента TComboBoxEx представлены в табл. 17.46.

Таблица 17.46. Методы компонента TComboBoxEx

Метод	Описание
procedure AddItem(Item: String; AObject: TObject);	Добавляет новый элемент и связанные с ним данные
procedure Clear; override ;	Очищает список
procedure ClearSelection; override ;	Проверяет, выделен ли какой-то элемент списка и, если так, снимает выделение (помещает -1 в свойство ItemIndex и очищает свойство Text)
procedure CopySelection (Destination: TCustomListControl); override ;	Проверяет, выделен ли какой-то элемент списка и, если так, копирует его в список Destination
procedure DeleteSelected; override ;	Проверяет, выделен ли какой-то элемент списка и, если так, удаляет его из списка
function Focused: Boolean; override ;	Возвращает значение True, если компонент имеет фокус ввода

Метод	Описание
procedure MoveSelection (Destination:	Проверяет, выделен ли какой-то элемент списка и, если так, копирует его в список Destination и затем удаляет
procedure SelectAll; override ;	Выделяет весь текст выбранного элемента. Чтобы выделить часть текста, используйте свойства SelStart и SelLength

События компонента TComboBoxEx перечислены в табл. 17.47.

Таблица 17.47. События компонента TComboBoxEx

Событие	Описание
property OnBeginEdit: TNotifyEvent;	Возникает, когда пользователь начинает изменять текст в поле списка
property OnChange: TNotifyEvent;	Возникает при каждом изменении пользователем (но не программой!) свойства Text
property OnDropDown: TNotifyEvent;	Возникает в момент раскрытия списка
property OnEndEdit: TNotifyEvent;	Возникает, когда пользователь заканчивает изменять текст в поле списка
property OnSelect: TNotifyEvent;	Возникает, когда пользователь выделяет элемент списка

Глава 18

Компоненты вкладки System

В этой главе описаны не все компоненты, представленные на вкладке **System** палитры компонентов, в частности, не описаны компоненты `TDDEClient`, `TDDEClientItem`, `TDDEServer` и `TDDEServerItem`. Эти компоненты рассчитаны на поддержку устаревшей технологии динамического обмена данными (Dynamic Data Exchange, DDE) между программами, которая в современных программах полностью вытеснена технологией связывания и внедрения объектов (Object Linking and Embedding, OLE).

TTimer — таймер

Компонент `TTimer` (таймер) служит для отсчета интервалов реального времени. Его свойство `Interval` определяет интервал времени в миллисекундах, который должен пройти от включения таймера до наступления события `OnTimer`. Таймер включается при установке значения `True` в его свойство `Enabled`. Однажды включенный таймер все время будет возбуждать события `OnTimer` до тех пор, пока его свойство `Enabled` не примет значения `False`.

Следует учесть, что в силу специфики реализации стандартного аппаратного таймера IBM-совместимого компьютера минимальный реально достижимый интервал отсчета времени в операционных системах MS-DOS, Windows 3.x и Windows 95/98/ME не может быть меньше 55 миллисекунд (этот интервал называется *тиком*). Более того, любой интервал времени, отсчитываемый с помощью таймера, в этих операционных системах всегда кратен 55 миллисекундам. Замечу, что для операционных систем Windows NT Server (Workstation), Windows 2000, Windows XP длительность тика составляет 10 мс. Чтобы убедиться в этом, проделайте следующий эксперимент (проект `Chap_18\Timer\Timer.dpr`), в котором подсчитывается среднее время между двумя срабатываниями таймера.

1. Начните новый проект с тестовой формой и поместите на нее компонент `TTimer`.
2. Установите в свойство `Enabled` таймера значение `False`.
3. Напишите модуль главной формы, представленный в листинге 18.1.

Листинг 18.1. Модуль главной формы программы, иллюстрирующей работу таймера¹

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons, ExtCtrls;
type
  TfmExample = class (TForm)
    Panel1: TPanel;
    bbRun: TBitBtn;
    bbClose: TBitBtn;
    edInput: TEdit;
    lbOutput: TLabel;
    mmOutput: TMemo;
    Timer1: TTimer;
    procedure bbRunClick(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure FormActivate(Sender: TObject);
  private
    { Private declarations }
    BegTime: TDateTime; // Начальное время цикла
    Counter: Integer;    // Счетчик цикла
  public
    { Public declarations }
  end;

var
  fmExample: TfmExample;

implementation

{$R *.DFM}

procedure TfmExample.bbRunClick(Sender: TObject);
// Запускает таймер. edInput содержит период его
// срабатывания.
var
  Delay: Word;
begin
  // Проверяем задание интервала

```

¹ Все представленные в книге тексты программ можно загрузить с сайта издательства по адресу <http://www.piter.com/download> — Примеч. ред.

```

if edInput.Text='' then Exit;
try
    Delay := StrToInt(edInput.Text);
except
    ShowMessage('Ошибка в записи числа');
    edInput.SelectAll;
    edInput.SetFocus;
    Exit
end;

Counter := 0;                                // Сбрасываем счетчик
Timer1.Interval := Delay;                    // Устанавливаем интервал
BegTime := Time;                             // Засекаем время
Timer1.Enabled := True;                       // Пускаем таймер
Screen.Cursor := crHourGlass
end;

procedure TfmExample.Timer1Timer(Sender: TObject);
var
    h, m, s, ms: Word; // Переменные для декодирования
                        // времени
const
    MaxCount = 55;      // Количество срабатываний таймера
begin
    Counter := Counter + 1;      // Нарастиваем счетчик
    срабатываний
    if Counter=MaxCount then      // Конец цикла?
    begin                          // - Да
        Timer1.Enabled := False; // Останавливаем таймер
        // Находим среднее время срабатывания:
        DecodeTime((Time-BegTime)/MaxCount, h, m, s, ms);
        mmOutput.Lines.Add(      // Выводим результат
            Format('Задано %s ms. Получено %d ms.',
                [edInput.Text, ms]));
        edInput.Text := '';      // Готовим следующий запуск
        edInput.SetFocus;
        Screen.Cursor := crDefault
    end;
end;

procedure TfmExample.FormActivate(Sender: TObject);
begin
    edInput.SetFocus
end;

end.

```

Необходимость нескольких (MaxCount) срабатываний для точного усреднения результата связана с тем, что системные часы обновляются каждые 55 мс (10 — для операционных систем на основе ядра NT). Вид окна работающей программы показан на рис. 18.1.

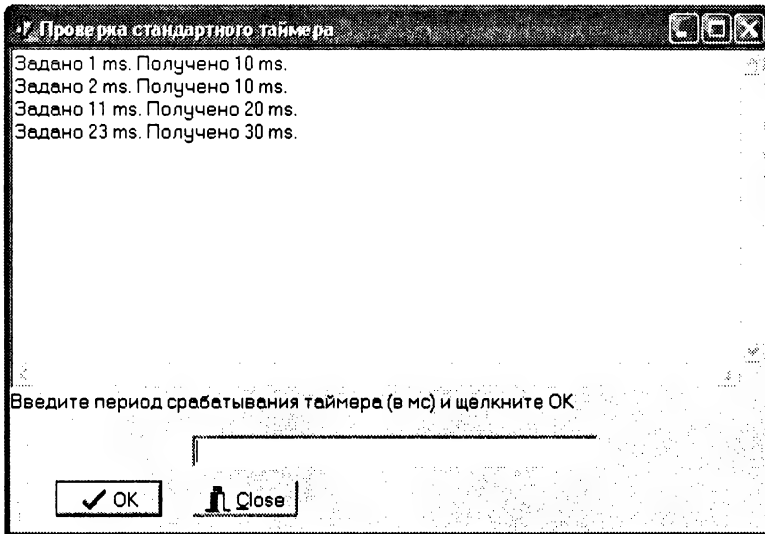


Рис. 18.1. Окно программы проверки таймера

TPaintBox — окно для рисования

Назначение компонента TPaintBox — предоставить пользователю простое окно с канвой для рисования произвольных изображений. Канва содержится в свойстве Canvas компонента, графические инструменты — в свойствах Font, Pen и Brush, а собственно рисование осуществляется в обработчике события OnPaint. Особенности использования этих инструментов описаны в разделе «Графический инструментарий» главы 13. Например, представленный ниже обработчик (проект Source\Chap_18\PaintBox\PaintDemo.dpr) создаст окно, показанное на рис. 18.2.

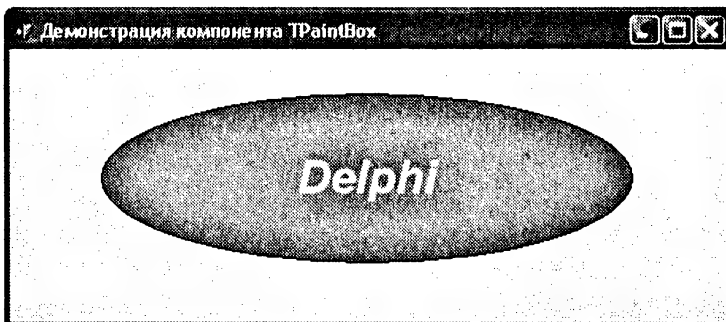


Рис. 18.2. Пример использования компонента TPaintBox


```

procedure TForm1.PaintBox1Paint(Sender: TObject);
var
    X, Y: Integer;
begin
    with PaintBox1, Canvas do
        begin
            Brush.Color := clRed;
            Ellipse(0, 0, Width, Height);
            Font.Name := 'Arial';
            Font.Size := Height div 5;
            Font.Style := [fsBold, fsItalic];
            Font.Color := clWhite;
            X := (Width - TextWidth('Delphi')) div 2;
            Y := (Height - TextHeight('D')) div 2;
            TextOut(X, Y, 'Delphi')
        end
    end;

```

TMediaPlayer — медиаплеер

Компонент TMediaPlayer представляет собой полнофункциональное мультимедийное устройство, способное, в частности, воспроизводить одновременно и изображение, и звук AVI-файлов. Чтобы увидеть компонент в действии, выполните следующую последовательность действий (проект Chap_18\MediaPlayer\MP.dpr).

1. Поместите на пустую форму две панели. Удалите содержимое их свойств Caption. Для нижней панели установите Align=alBottom, для верхней — Align=alClient.
2. Верхняя панель будет экраном для отображения изображения. Установите для нее Name=pnDisplay.
3. Поместите на нижнюю панель компонент TMediaPlayer и введите в его свойство Display ссылку на панель pnDisplay. Раскройте список свойства FileName и выберите в нем файл c:\Program Files\Borland\Delphi7\Demos\CoolStuff\speedis.avi¹. Поместите в его свойство AutoOpen значение True.
4. Напишите такой обработчик события OnResize для формы:

```

procedure TForm1.DisplayResize(Sender: TObject);
begin
    // Устанавливаем границы изображения
    MP.DisplayRect :=
        Rect(0, 0, Display.Width, Display.Height);
end;

```

Если ваш компьютер оснащен звуковой картой, включите звуковые колонки и запустите программу. После щелчка мышью на кнопке  вы увидите велико-

¹ Маршрут доступа c:\Program Files\Borland\Delphi7 указан для умалчиваемого расположения Delphi. Если система установлена в другую папку, этот маршрут нужно скорректировать.

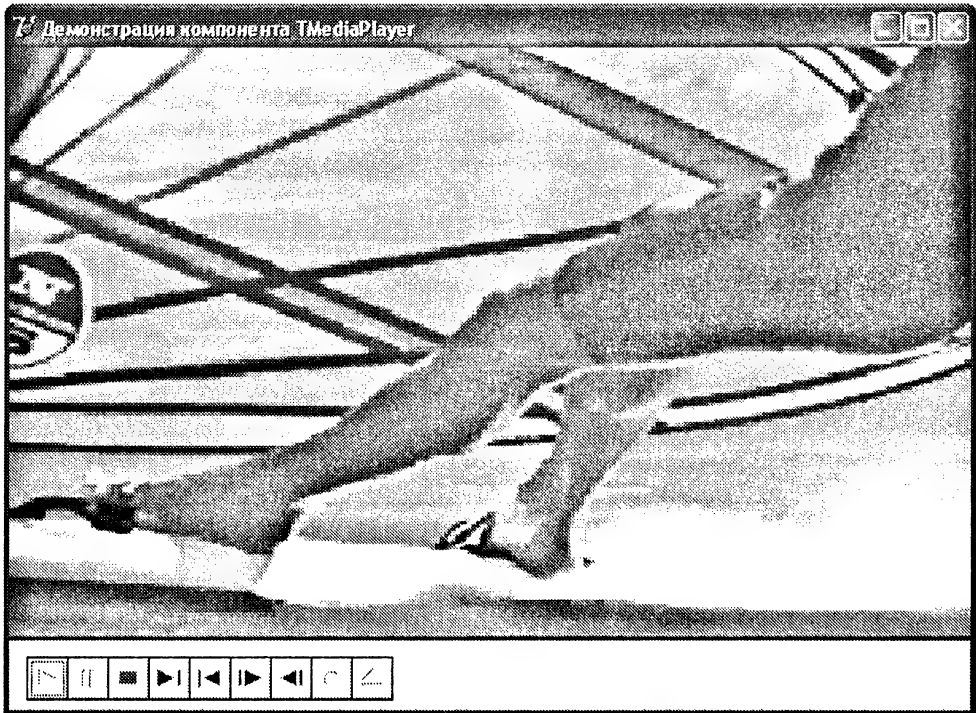


Рис. 18.3. Демонстрация компонента TMediaPlayer

лепный рекламный ролик «Speed is Delphi». Причем вы можете изменять размеры окна программы и таким способом получить полноэкранное видео (рис. 18.3).

Как видите, использование компонента предельно просто. Он автоматически распознает тип мультимедийного устройства по расширению файла и берет на себя управление этим устройством. Разумеется, в каждый момент времени компонент может управлять лишь одним устройством, однако опосредованно путем изменения содержимого свойства `FileName` или явно с помощью свойства `DeviceType` программа может менять устройство, связанное с компонентом.

Каждая кнопка компонента имеет собственное имя, позволяющее программисту сделать какую-либо кнопку невидимой или недоступной. В табл. 18.1 приводятся имя и назначение каждой кнопки (под *носителем информации* подразумеваются файлы и физические устройства, которые могут служить источником или приемником информации).

Таблица 18.1. Назначение кнопок компонента TMediaPlayer

Кнопка	Имя	Описание
	<code>btPlay</code>	Иницирует воспроизведение носителя информации
	<code>btPause</code>	Прерывает процесс записи или воспроизведения информации. Повторный щелчок на этой кнопке возобновляет прерванный процесс

— продолжение ↗

Таблица 18.1 (продолжение)

Кнопка	Имя	Описание
	btStop	Останавливает запись или воспроизведение
	btNext	Позиционирует устройство на следующую дорожку или в конец носителя информации, если устройство не имеет дорожек
	btPrev	Позиционирует устройство на предыдущую дорожку или в начало носителя информации, если устройство не имеет дорожек
	btStep	Позиционирует устройство на один блок кадров ближе к концу носителя информации. Количество кадров в блоке содержится в свойстве Frames компонента
	btBack	Позиционирует устройство на один блок кадров ближе к началу носителя информации
	btRecord	Начинает запись в носитель информации
	btEject	Освобождает устройство от носителя информации

Свойства компонента TMediaPlayer представлены в табл. 18.2.

Таблица 18.2. Свойства компонента TMediaPlayer

Свойство	Описание
property AutoEnable: Boolean;	Разрешает/запрещает автоматическое определение доступных и недоступных кнопок
property AutoOpen: Boolean;	Разрешает/запрещает автоматический старт записи/воспроизведения в момент создания формы, в которую вставлен компонент
property AutoRewind: Boolean;	Разрешает/запрещает автоматическую «перемотку» носителя информации в момент начала записи или воспроизведения. Игнорируется, если устройство имеет дорожки или если установлены значения в свойства StartPos и EndPos
TMPDevCaps = (mpCanStep, mpCanEject, mpCanPlay, mpCanRecord, mpUsesWindows); TMPDevCapsSet = set of TMPDevCaps; property Capabilities: TMPDevCapsSet;	Содержит параметры устройства, связанного с компонентом: mpCanStep — может смещать носитель на кадр вперед или назад (устройство типа Animation, AVI Video, Digital Video, Overlay или VCR); mpCanEject — может автоматически удалять носитель из устройства; mpCanPlay — может воспроизводить информацию; mpCanRecord — может

Свойство	Описание
	записывать информацию; mpUsesWindows — может использовать окно для отображения процесса
TMPBtnType = (btPlay, btPause, btStop, btNext, btPrev, btStep, btBack, btRecord, btEject); TButtonSet = set of TMPBtnType; property ColoredButtons: TButtonSet;	Кнопки, перечисленные в этом свойстве, будут использовать цвета, заданные по умолчанию (не перечисленные кнопки будут черно-белыми). По умолчанию множество ColoredButtons содержит все кнопки
property DeviceID: Word;	Содержит Windows-идентификатор устройства, связанного с компонентом. Доступно только для чтения
TMPDeviceTypes = (dtAutoSelect, dtAVIVideo, dtCDAudio, dtDAT, dtDigitalVideo, dtMMMovie, dtOther, dtOverlay, dtScanner, dtSequencer, dtVCR, dtVideodisc, dtWaveAudio); property DeviceType: TMPDeviceTypes;	Содержит тип устройства. Если DeviceType=dtAutoSelect, тип определяется автоматически по расширению файла в свойстве FileName. В момент установки любого мультимедийного устройства в файл SYSTEM.INI заносятся расширения файлов, которые поддерживаются этим устройством
property Display: TWinControl;	Содержит ссылку на окно для устройств, которые могут его использовать в процессе записи/воспроизведения
property DisplayRect: TRect;	Содержит область окна, указанного в свойстве Display, которое устройство может использовать для отображения процесса записи/воспроизведения. Для улучшения процесса отображения рекомендуется в свойстве DisplayRect указывать размеры, заданные по умолчанию (координаты 0, 0 для правого нижнего угла)
property EnabledButtons: TButtonSet;	Содержит множество доступных кнопок (тип TButtonSet описан выше для свойства ColoredButtons)
property EndPos: LongInt;	Содержит положение носителя информации в момент прекращения записи или воспроизведения. Переустановка значения приводит к повторению записи или воспроизведения
property Error: LongInt;	Содержит код ошибки. Доступно только для чтения
property ErrorMessage: String;	Содержит сообщение об ошибке. Доступно только для чтения
property FileName: String;	Содержит имя читаемого или записываемого файла

— продолжение ➤

Таблица 18.2 (продолжение)

Свойство	Описание
property Frames: LongInt;	Определяет количество кадров, пропускаемых при выполнении методов Step или Back
property Length: LongInt;	Содержит текущую длину носителя информации. Доступно только для чтения
TMPModes = (mpNotReady, mpStopped, mpPlaying, mpRecording, mpSeeking, mpPaused, mpOpen); property Mode: TMPModes;	Показывает текущее состояние мультимедийного устройства: mpNotReady — не готово; mpStopped — остановлено; mpPlaying — воспроизводит информацию; mpRecording — записывает информацию; mpSeeking — перемещает носитель информации; mpPaused — приостановлено; mpOpen — открыто. Доступно только для чтения
property Notify: Boolean;	Содержит значение True, если методы Back, Close, Eject, Next, Open, Pause, PauseOnly, Play, Previous, StartRecording, Resume, Rewind, Step и Stop будут генерировать событие OnNotify
TMPNotifyValues = (nvSuccessful, nvSuperseded, nvAborted, nvFailure); property NotifyValue: TMPNotifyValues;	Содержит результат последней команды, генерирующей событие OnNotify: nvSuccessful — выполнена успешно; nvSuperseded — заменена другой командой; nvAborted — досрочно прервана; nvFailure — завершилась ошибкой. Доступно только для чтения
property Position: LongInt;	Содержит текущую позицию носителя информации
property Shareable: Boolean;	Разрешает/запрещает другим компонентам и программам использовать устройство, связанное с данным компонентом
property Start: LongInt;	Содержит позицию от начала носителя, с которой начинается запись или воспроизведение. Определяется в момент открытия устройства и доступно только для чтения
property StartPos: LongInt;	Содержит текущую позицию от начала носителя, с которой начинается запись или воспроизведение
TMPTimeFormats = (tfMilliseconds, tfHMS, tfMSF, tfFrames, tfSMPTE24, tfSMPTE25, tfSMPTE30, tfSMPTE30Drop, tfBytes, tfSamples, tfTMSF);	Определяет формат представления данных в свойствах StartPos, Length, Position, Start и EndPos (см. пояснения ниже)

Свойство	Описание
property TimeFormat: TMTTimeFormats;	
property TrackLength[TrackNum: Integer]: LongInt;	Возвращает длину (в формате TimeFormat) дорожки с индексом TrackNum. Доступно только для чтения
property TrackPosition [TrackNum: Integer]: LongInt	Возвращает стартовую позицию (в формате TimeFormat) дорожки с индексом TrackNum. Доступно только для чтения
property Tracks: LongInt;	Содержит количество дорожек в открытом устройстве. Доступно только для чтения
property VisibleButtons: TButtonSet;	Содержит множество видимых кнопок (тип TButtonSet описан выше для свойства ColoredButtons)
property Wait: Boolean;	Если содержит значение True, управление возвращается в программу только после завершения перехода устройства в другое состояние, связанное с методом Back, Close, Eject, Next, Open, Pause, PauseOnly, Play, Previous, StartRecording, Resume, Rewind, Step или Stop

Если свойство AutoEnable имеет значение True, компонент автоматически по типу устройства или расширению файла и значению свойства Mode определяет, какие кнопки будут доступны пользователю. Если свойство AutoEnable имеет значение False, доступными будут только те кнопки, которые входят во множество EnabledButtons. Множество EnabledButtons игнорируется, если свойство AutoEnable имеет значение True.

Свойство AutoRewind игнорируется, если устройство имеет дорожки или если процесс записи/воспроизведения вызван изменением значения в свойстве StartPos или EndPos.

Положение носителя в свойствах StartPos, Length, Position, Start и EndPos в зависимости от значения свойства TimeFormat должно интерпретироваться следующим образом (отсчет ведется от начала носителя информации):

- tfMilliseconds — миллисекунды в диапазоне от 0 до 4 294 967 295;
- tfHMS — часы, минуты, секунды (занимают в порядке перечисления по одному байту начиная с младшего; старший байт LongInt не используется);
- tfMSF — минуты, секунды и кадры (занимают в порядке перечисления по одному байту начиная с младшего; старший байт LongInt не используется);
- tfFrames — кадры в диапазоне от 0 до 4 294 967 295;
- tfSMPTE24 — часы, минуты, секунды и количество блоков по 24 кадра в секунду (занимают в порядке перечисления по одному байту, начиная с младшего);
- tfSMPTE25 — часы, минуты, секунды и количество блоков по 25 кадров в секунду;

- `tfSMPTE30` — часы, минуты, секунды и количество блоков по 30 кадров в секунду;
- `tfSMPTE30Drop` — часы, минуты, секунды и количество пропущенных блоков по 30 кадров в секунду;
- `tfBytes` — байты в диапазоне от 0 до 4 294 967 295.
- `tfSamples` — количество условных блоков информации в диапазоне от 0 до 4 294 967 295;
- `tfTMSF` — дорожки, минуты, секунды и кадры (занимают в порядке перечисления по одному байту, начиная с младшего).

Свойство `TimeFormat` в окне инспектора объектов недоступно и устанавливается автоматически при открытии мультимедийного устройства методом `Open`. В связи с этим значения в свойства `StartPos`, `Position` и `EndPos` могут устанавливаться, а в свойствах `Length`, `Start`, `TrackLength` и `TrackPosition` становятся действительными только после вызова метода `Open`.

Методы компонента `TMediaPlayer` перечислены в табл. 18.3.

Таблица 18.3. Методы компонента `TMediaPlayer`

Метод	Описание
procedure <code>AutoButtonSet</code> (<code>Btn: TMPBtnType</code>); dynamic ;	Реализует доступность/недоступность кнопок компонента при установке в свойство <code>AutoEnable</code> значения <code>True</code>
procedure <code>Back</code> ;	Перемещает текущую позицию в носителе информации назад на количество кадров, содержащихся в свойстве <code>Frames</code>
procedure <code>Click</code> (<code>Button: TMPBtnType</code> ; var <code>DoDefault: Boolean</code>); dynamic ;	По умолчанию вызывает обработчик события <code>OnClick</code> . Метод должен перекрываться в потомках
procedure <code>Close</code> ;	Закрывает связанное с компонентом устройство
procedure <code>DoNotify</code> ; dynamic ;	По умолчанию вызывает обработчик события <code>OnNotify</code> . Метод должен перекрываться в потомках
procedure <code>Eject</code> ;	Выгружает носитель информации из устройства
procedure <code>MMNotify</code> (var <code>Message: TMessage</code>); message <code>MM_MCINOTIFY</code> ;	Реализует отклик на сообщение <code>Windows MM_MCINOTIFY</code> . По умолчанию вызывает метод <code>DoNotify</code>
procedure <code>Next</code> ;	Перемещает текущую позицию носителя на начало следующей дорожки, а если устройство не имеет дорожек — в конец носителя
procedure <code>Notification</code> (<code>AComponent: TComponent</code> ; <code>Operation: TOperation</code>); override ;	Извещает компонент <code>AComponent</code> о том, что завершилась операция <code>Operation</code>

Метод	Описание
procedure Open;	Открывает мультимедийное устройство
procedure Pause;	Приостанавливает процесс воспроизведения или записи информации. Если устройство уже было приостановлено, восстанавливает процесс, вызывая метод Resume
procedure PauseOnly;	Приостанавливает процесс воспроизведения или записи информации. Если устройство уже было приостановлено, ничего не делает
procedure Play;	Начинает воспроизведение с позиции startPos, а если это свойство не установлено — с текущей позиции Position
procedure PostClick (Button: TMPBtnType); dynamic ;	Реализует отклик на событие OnPostClick. По умолчанию вызывает соответствующий обработчик события
procedure Previous;	Перемещает текущую позицию носителя на начало предыдущей дорожки, а если устройство не имеет дорожек — в начало носителя
procedure Resume;	Восстанавливает процесс записи или воспроизведения. Вызывается при повторном щелчке на кнопке btPause
procedure Rewind;	Перемещает текущую позицию носителя к положению, определяемому свойством Start
procedure Save;	Сохраняет информацию в файле с именем FileName
procedure StartRecording;	Начинает запись информации с текущей позиции или позиции startPos
procedure Step;	Перемещает текущую позицию в носителе информации вперед на количество кадров, содержащихся в свойстве Frames
procedure Stop;	Прекращает запись или воспроизведение информации

События компонента TMediaPlayer представлены в табл. 18.4.

Таблица 18.4. События компонента TMediaPlayer

Событие	Описание
EMPNotify = procedure (Sender: TObject; Button: TMPBtnType; var DoDefault:	Возникает после щелчка на кнопке Button компонента. Если DoDefault=True, вызывается метод, связанный с этой кнопкой

— продолжение ➤

Таблица 18.4 (продолжение)

Событие	Описание
Boolean) of object; property OnClick: EMPNotify;	
property OnNotify: TNotifyEvent;	Возникает после завершения вызова одного из методов Back, Close, Eject, Next, Open, Pause, PauseOnly, Play, Previous, Resume, Rewind, StartRecording, Step или Stop, если свойство Notify имеет значение True
EMPPostNotify = procedure (Sender: TObject; Button: TMPBtnType) of object; property OnPostClick: EMPPostNotify;	Если Wait=True, событие OnPostClick возникает только после завершения соответствующей операции, вызванной событием OnClick, в противном случае — немедленно после события OnClick

Иногда требуется воспроизвести короткий звуковой сигнал для привлечения внимания пользователя программы — например, после завершения длительного по времени процесса обновления данных. Стандартная для Delphi процедура без параметров Beep извлекает сигнал из системного динамика только в том случае, если компьютер не оснащен звуковой картой. С другой стороны, мощные возможности компонента TMediaPlayer кажутся излишними для решения этой задачи на компьютерах, оснащенных звуковой картой. В этом случае может пригодиться API-функция MessageBeep, с помощью которой в 32-разрядных версиях Windows озвучиваются стандартные диалоговые окна. Единственным параметром обращения к этой функции является один из следующих идентификаторов стандартного звука: mb_IconAsterisk, mb_IconExclamation, mb_IconHand, mb_IconQuestion и mb_Ok. Обращение с параметром \$FFFFFFFF игнорирует звуковую карту и извлекает короткий звуковой сигнал из системного динамика. Сразу после обращения функция возвращает управление программе и воспроизводит звук асинхронно.

Значительно более богатые возможности в этом отношении имеет API-функция PlaySound, которая способна воспроизводить любые звуковые клипы.

Объявление функции:

```
function PlaySound(pszSound: PChar; hmod: HINST;  
    fdwSound: Cardinal): Boolean; stdcall; external  
    'winmm.dll';
```

Параметры функции перечислены ниже:

- pszSound — имя воспроизводимого файла;
- hmod — дескриптор программы, если звук берется из ресурсного файла (в противном случае содержит 0);
- fdwSound — параметр, уточняющий воспроизведение; его значениями могут быть:
 - SND_ASYNC — звук воспроизводится асинхронно (функция возвращает управление сразу после обращения к ней);

- ❑ SND_LOOP — звук воспроизводится постоянно до тех пор, пока не будет вызвана та же функция с параметром `pszSound=0`;
- ❑ SND_NOSTOP — функция пытается воспроизвести звук, если устройство не занято; в противном случае не воспроизводит ничего (если устройство занято, но этот параметр не указан, функция прервет воспроизведение предыдущего звука и начнет воспроизведение следующего);
- ❑ SND_NOWAIT — если устройство занято, функция не будет воспроизводить новый звук;
- ❑ SND_PURGE — прекращает воспроизведение всех звуков для данной задачи;
- ❑ SND_SYNC — воспроизводит звук синхронно и возвращает управление только после окончания его воспроизведения.

TOleContainer — контейнер OLE-объекта

Компонент `TOleContainer` является удобным контейнером для размещения связанного или внедренного OLE-объекта (Object Linked or Embedding — связывание или внедрение объекта). Такие объекты (таблицы, картинки, фрагменты текста и т. п.) на форме Delphi-программы выглядят как обычно или заменяются значками. Замечательной особенностью OLE-объекта является то, что его активизация (обычно двойным щелчком мышью) приводит к активизации связанной с объектом программы, которая называется OLE-сервером и которая после загрузки показывает на экране свое окно и предоставляет пользователю средства редактирования объекта.

Типичными OLE-серверами являются такие системные утилиты, как `Paint` и `Notepad`, текстовый процессор `Word`, табличный процессор `Excel` и др.

Чтобы наглядно продемонстрировать возможности компонента, проделайте следующее упражнение (проект `Chap_18\OleContainer\OLEDemo.dpr`).

1. Создайте новое приложение с пустой формой и вставьте в нее главное меню `TMainMenu` с единственной командой `Файл`.
2. Поместите на форму компонент `TOleContainer`. Установите в его свойство `Align` значение `alClient`.
3. Напишите такой обработчик события `OnCreate` для формы `Form1`:

```
procedure TForm1.FormCreate(Sender: TObject);
const
    BMP_Name =
        'C:\Program Files\Common Files\Borland '+
        'Shared\Images\Splash\16Color\athena.bmp';
begin
    OleContainer1.CreateObjectFromFile(BMP_Name, False)
end;
```

Запустите программу и дважды щелкните на изображении мышью. После паузы, вызванной загрузкой программы-сервера, вы увидите окно, показанное на рис. 18.4.



Рис. 18.4. Пример связывания OLE-объекта с сервером Paint

Теперь вы можете использовать средства графического редактора Paint для редактирования изображения. Чтобы завершить редактирование, нажмите клавишу Esc — окно примет первоначальный вид, но изображение в нем сохранит все внесенные в него изменения.

Не правда ли, весьма впечатляющая демонстрация мощных возможностей компонента: ведь для полноценного доступа к сложной технологии OLE понадобился единственный вызов одного из его методов.

Компонент может создавать как связанные, так и внедренные объекты и открывать окно сервера как в отдельном окне, так и в окне содержащей его программы-клиента. В последнем случае при запуске сервера происходит автоматическая замена пунктов главного меню программы с групповыми индексами 1, 3 и 5 (если они есть — вот почему в примере вставлено главное меню с недействующей командой Файл) пунктами главного меню сервера, а также вставка в окно программы инструментальных панелей сервера. Если компонент открывается в окне клиента, его свойство Align (или такое же свойство панели, на которую он помещен) должно иметь значение alClient.

Свойства, методы и события компонента TOleContainer представлены в табл. 18.5, 18.6 и 18.7.

Таблица 18.5. Свойства компонента TOleContainer

Свойство	Описание
property AllowInPlace:	Определяет способ размещения OLE-сервера. Если имеет значение True и свойство Icon=False, окну сервера с редактируемым объектом размещается в пределах компонента, в противном случае создается отдельное окно для сервера (независимо от значения Icon)

Свойство	Описание
TAutoActivate = (aaManual, aaGetFocus, aaDoubleClick); property AutoActivate: TAutoActivate;	Определяет способ активизации OLE-объекта: aaManual — активизируется путем программного вызова метода DoVerb (ovShow); aaGetFocus — активизируется при получении фокуса ввода; aaDoubleClick — активизируется при двойном щелчке мышью
property AutoVerbMenu: Boolean;	Если имеет значение True, для компонента автоматически создается вспомогательное меню, содержащее доступные команды OLE-сервера
property CanPaste: Boolean;	Имеет значение True, если буфер обмена (clipboard) содержит OLE-объект (или связь с ним), который(ую) можно вставить в компонент. Доступно только для чтения
property CopyOnSave: Boolean;	Если содержит True, перед сохранением объекта в файле или потоке данных он предварительно сжимается для экономии размеров файла (потока данных). Если в компоненте размещен очень большой объект и динамической памяти может не хватить для создания временной копии сжатого объекта, в свойство следует поместить значение False
property Iconic: Boolean;	Содержит True, если объект будет заменяться значком OLE-сервера, и False, если объект изображается так, как он появится в окне сервера
property Linked: Boolean;	Содержит True, если объект связан с программой, и False — если является ее частью (внедрен в нее). Доступно только для чтения
property Modified: Boolean;	Содержит True, если объект был изменен (в том числе заменен другим или уничтожен)
property NewInserted: Boolean;	Содержит True, если объект был заново создан с помощью метода InsertObjectDialog. В этом случае вызывайте метод DoVerb (ovShow), чтобы показать объект в окне сервера. Доступно только для чтения
property ObjectVerbs: TStrings;	Содержит список команд, которые программа может посылать серверу. Доступно только для чтения
property OldStreamFormat: Boolean;	В это свойство следует поместить True перед чтением объекта из файла или потока данных, если объект был создан версией Delphi 1
property OleClassName: String;	Содержит имя класса, под которым (именем) был зарегистрирован OLE-сервер в реестре Windows. Доступно только для чтения
property OleObject: Variant;	Содержит ссылку на OLE-объект

Таблица 18.5 (продолжение)

Свойство	Описание
property PrimaryVerb: Integer;	Содержит индекс умалчиваемой команды для OLE-объекта
TSizeMode = (smClip, smCenter, smScale, smStretch, smAutoSize); property SizeMode: TSizeMode;	Определяет способ размещения OLE-объекта в контейнере: smClip — отсекаются любые части объекта, выходящие за границы контейнера; smCenter — центрируется в контейнере; smScale — масштабируется так, чтобы целиком заполнить границы контейнера; smStretch — масштабируется так, чтобы не нарушились пропорции объекта; smAutoSize — изменяются границы контейнера, чтобы полностью отобразить объект без искажений
property SourceDoc: String;	Содержит имя файла для связанного объекта
type TObjectState = (osEmpty, osLoaded, osRunning, osOpen, osInPlaceActive, osUIActive); property State: TObjectState;	Определяет состояние OLE-объекта: osEmpty — контейнер не содержит объект; osLoaded — объект загружен, но его сервер не запущен; osRunning — запущен OLE-сервер; osOpen — объект открыт в отдельном окне; osInPlaceActive — объект открыт в окне контейнера, но сервер еще не заменил команды меню и инструментальные панели программы на свои (промежуточное состояние перед osUIActive); osUIActive — объект открыт в окне контейнера

Таблица 18.6. Методы компонента TOleContainer

Метод	Описание
procedure Close;	Закрывает сервер. Все сделанные к этому моменту изменения в объекте автоматически сохраняются
procedure Copy;	Копирует объект в буфер обмена
procedure CreateLinkToFile(const FileName: String; Iconic: Boolean);	Создает связанный объект по имени файла FileName. Параметр Iconic содержит True, если изображение объекта заменяется значком
procedure CreateObject(const OleClassName: String; Iconic: Boolean);	Создает внедренный объект по имени класса сервера OleClassName. Параметр Iconic содержит True, если изображение объекта заменяется значком
procedure CreateObjectFromFile(const FileName: String; Iconic: Boolean);	Создает внедренный объект по имени файла FileName. Параметр Iconic содержит True, если изображение объекта заменяется значком

Метод	Описание
procedure DestroyObject;	Уничтожает объект без сохранения сделанных в нем изменений
procedure DoVerb(Verb: Integer);	Требует от объекта выполнить действие с индексом Verb из списка ObjectVerbs
function GetIconMetaPict: HGlobal;	Возвращает указатель на распределенный в куче метафайл для прорисовки значка объекта на другой канве
function InsertObjectDialog: Boolean;	Создает и показывает диалоговое окно создания объекта. Возвращает True, если пользователь закрыл окно щелчком на кнопке ОК
procedure LoadFromFile(const FileName: String);	Загружает объект из файла FileName
procedure LoadFromStream(Stream: TStream);	Загружает объект из потока данных Stream
function ObjectPropertiesDialog: Boolean;	Создаст и показывает диалоговое окно изменения свойств объекта. Возвращает True, если пользователь закрыл окно щелчком на кнопке ОК
procedure Paste;	Создает объект по данным, хранящимся в буфере обмена
function PasteSpecialDialog: Boolean;	Создаст и показывает диалоговое окно специальной вставки объекта. Возвращает True, если пользователь закрыл окно щелчком на кнопке ОК
procedure Run;	Запускает сервер, но не активизирует объект. Активизация объекта при запущенном сервере происходит значительно быстрее
procedure SaveAsDocument(const FileName: String);	Сохраняет объект в файле FileName с использованием формата OLE Document
procedure SaveToFile(const FileName: String);	Сохраняет объект в файле FileName
procedure SaveToStream(Stream: TStream);	Сохраняет объект в потоке данных Stream
procedure UpdateObject;	Обновляет объект, вновь загружая его из источника
procedure UpdateVerbs;	Обновляет список ObjectVerbs (некоторые команды могут его изменить, например, команда Play может заменить Play на Stop)

Таблица 18.7. События компонента TOleContainer

Событие	Описание
property OnActivate: TNotifyEvent;	Возникает сразу после активизации объекта
property OnDeactivate: TNotifyEvent;	Возникает сразу после перехода объекта в неактивное состояние

Глава 19

Компоненты вкладки Dialogs

В состав Windows входит ряд типовых диалоговых окон, таких как окно выбора загружаемого файла, выбора шрифта, настройки принтера и т. д. В Delphi реализованы классы, объекты которых дают программисту удобные способы создания и использования таких окон.

Работа со стандартными диалоговыми окнами осуществляется в три этапа.

Вначале на форму помещается соответствующий компонент и осуществляется настройка его свойств (следует заметить, что собственно компонент не виден в момент работы программы, видно лишь создаваемое им стандартное окно). Настройка свойств может проходить как на этапе конструирования, так и в ходе прогона программы. Как и для любых других компонентов, программист не должен заботиться о вызове конструктора и деструктора класса диалогового окна — эти вызовы реализуются автоматически в момент старта и завершения программы.

На втором этапе осуществляется вызов стандартного для классов диалоговых окон метода `Execute`, который создает и показывает на экране диалоговое окно. Вызов этого метода обычно располагается внутри обработчика какого-либо события. Например, обработчик выбора в меню команды Открыть файл может вызвать метод `Execute` компонента `TOpenDialog`, обработчик щелчка на кнопке Сохранить панели инструментов может вызвать такой же метод компонента `TSaveDialog` и т. д. Только после обращения к методу `Execute` на экране появляется соответствующее диалоговое окно. Это окно является модальным, поэтому сразу после обращения к методу `Execute` дальнейшее выполнение программы приостанавливается до тех пор, пока пользователь не закроет окно. Поскольку `Execute` — логическая функция, она возвращает в программу значение `True`, если результат диалога с пользователем был успешным.

Проанализировав результат вызова метода `Execute`, программа может выполнить третий этап — использовать введенные с помощью диалогового окна данные — имя файла, параметры принтера, выбранный шрифт и т. д.

Проиллюстрируем сказанное следующим примером. Создадим простую программу для просмотра содержимого текстового файла. Для этого на пустую форму

поместите компонент `TOpenDialog`, а также кнопку `TButton` и многострочное поле `TMemo`. При работе программы щелчок на кнопке будет сигналом о необходимости загрузить в поле новый файл. Создайте представленный ниже обработчик события `OnClick` этой кнопки:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    // Настраиваем компонент на отбор текстовых файлов:
    OpenDialog1.Filter := 'Текстовые файлы|*.txt|'+
                        'Файлы Паскаля|*.pas';
    // Выполняем диалог и анализируем его результат
    if OpenDialog1.Execute and
    FileExists(OpenDialog1.FileName)
    then // Результат успешный - пользователь выбрал файл.
        // Загружаем файл в поле Memo1:
        Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
end;
```

TOpenDialog и TSaveDialog — окна открытия и сохранения файлов

Компоненты `TOpenDialog` и `TSaveDialog` имеют идентичные свойства и поэтому рассматриваются вместе. Пример диалогового окна `TOpenDialog` показан на рис. 19.1.

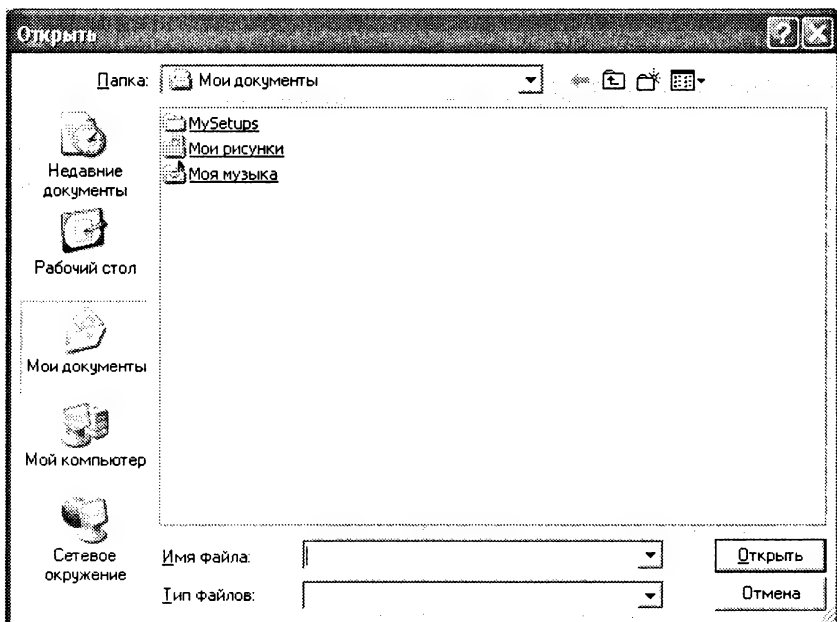


Рис. 19.1. Стандартное окно `TOpenDialog`

Свойство **FileName**: **String** содержит путь поиска и выбранный файл при успешном завершении диалога с пользователем. Программа может применять это свойство для доступа к файлу с целью читать из него данные (TOpenDialog) или записывать в него (TSaveDialog). Замечу, что пользователь может ввести произвольное имя и, следовательно, указать несуществующий файл. Для записи это не имеет значения, но при чтении отсутствие файла может привести к краху программы. Чтобы избежать этого, можно проверить существование файла глобальной функцией **FileExists**, как это сделано в предыдущем примере, или использовать механизм обработки исключительных ситуаций.

Свойство **Filter**: **String** используется для фильтрации (отбора) файлов, показываемых в диалоговом окне. Это свойство можно устанавливать с помощью специального редактора на этапе конструирования формы или программно, как это сделано в предыдущем примере. Для доступа к редактору достаточно щелкнуть на кнопке  в строке свойства **Filter** окна инспектора объектов. При программном вводе фильтры задаются одной длинной строкой, в которой символы вертикальной черты (|) служат для разделения фильтров, а также для отделения описания фильтруемых файлов от соответствующей маски выбора. Например, следующий оператор задает две маски для отбора файлов с расширениями **PAS** и **TXT**:

```
OpenDialog1.Filter :=
  'Текстовые файлы|*.txt|Файлы Паскаля|*.pas';
```

Установить начальный каталог позволяет свойство **InitialDir**: **String**. Например:

```
OpenDialog1.InitialDir :=
  'c:\program files\borland\delphi6\source';
```

С помощью свойства **DefaultExt**: **String** формируется полное имя файла, если при ручном вводе пользователь не указал расширение. В этом случае к имени файла прибавляется разделительная точка и содержимое этого свойства.

Настройка диалогового окна может варьироваться с помощью следующего свойства:

```
type TOpenOption = (ofReadOnly, ofOverwritePrompt,
  ofHideReadOnly, ofNoChangeDir, ofShowHelp, ofNoValidate,
  ofAllowMultiSelect, ofExtensionDifferent,
  ofPathMustExist, ofFileMustExist, ofCreatePrompt,
  ofShareAware, ofNoReadOnlyReturn, ofNoTestFileCreate,
  ofNoNetworkButton, ofNoLongNames, ofOldStyleDialog,
  ofNoDereferenceLinks, ofEnableIncludeNotify,
  ofEnableSizing, ofDontAddToRecent, ofForceShowHidden);
TOpenOptions = set of TOpenOption;
property Options: TOpenOptions;
```

Значения этого свойства имеют следующий смысл:

-  ofReadOnly — устанавливает флажок Только чтение;
-  ofOverwritePrompt — требует согласия пользователя при записи в существующий файл;
-  ofHideReadOnly — скрывает флажок Только чтение;

- ofNoChangeDir — запрещает смену каталога;
- ofShowHelp — включает в окно кнопку Справка;
- ofNoValidate — запрещает автоматическую проверку правильности набираемых в имени файла символов;
- ofAllowMultiSelect — разрешает множественный выбор файлов;
- ofExtensionDifferent — при завершении диалога наличие этого значения в свойстве Options говорит о том, что пользователь ввел расширение, отличающееся от заданного по умолчанию;
- ofPathMustExist — разрешает указывать файлы только из существующих каталогов;
- ofFileMustExist — разрешает указывать только существующие файлы;
- ofCreatePrompt — требует подтверждения для создания несуществующего файла;
- ofShareAware — разрешает выбирать файлы, используемые другими параллельно выполняемыми программами;
- ofNoReadOnlyReturn — запрещает выбор файлов, имеющих атрибут *только для чтения*;
- ofNoTestFileCreate — запрещает проверку доступности сетевого или локального диска;
- ofNoNetworkButton — запрещает вставку кнопки создания сетевого диска;
- ofNoLongNames — запрещает использование длинных имен файлов;
- ofOldStyleDialog — создает диалоговое окно в стиле Windows 3.x;
- ofNoDereferenceLinks — запрещает разыменовывание ярлыков Windows (если пользователь выбирает ярлык, в свойство FileName помещается путь доступа и имя файла ярлыка с расширением LNK, а не файла, на который указывает ярлык);
- ofEnableIncludeNotify (Windows 2000 и выше) — разрешает посылку уведомляющих сообщений диалоговому окну, если пользователь открывает папку (сообщения посылаются для каждого файла папки, что позволяет управлять отображением содержимого папки; см. также описание события OnIncludeItem);
- ofEnableSizing (Windows 98 и выше) — позволяет окну в стиле Проводника Windows изменять размеры (диалоговые окна в старом стиле изменение размеров не допускают) с помощью мыши или клавиатуры (по умолчанию такое изменение размеров допускается независимо от этого флага, который учитывается только в пользовательских диалоговых окнах);
- ofDontAddToRecent — запрещает присоединять файл к списку недавно открывавшихся файлов;
- ofShowHidden — показывает скрытые файлы.

Если разрешен множественный выбор, доступ к выбранным именам можно получить в свойстве Files: **Strings**.

События компонентов TOpenDialog и TSaveDialog перечислены в табл. 19.1.

Таблица 19.1. События компонентов TOpenDialog и TSaveDialog

Событие	Описание
type TCloseQueryEvent = procedure (Sender: TObject; var CanClose: Boolean) of object ; property OnCanClose: TCloseQueryEvent;	Возникает непосредственно перед закрытием диалогового окна. Используйте обработчик этого события для окончательной проверки правильности выбранного файла, например, его существования, возможности перезаписи содержимого и т. п. Если обработчик установит значение False в параметр CanClose, окно не будет закрыто
property OnClose: TNotifyEvent;	Возникает при закрытии диалогового окна
property OnFolderChange: TNotifyEvent;	Возникает при изменении отображаемой папки
type _OFNOTIFYEXA = packed record hdr: TNMHdr; lpOFN: POpenFilename; psf: IShellFolder; pidl: Pointer; end ; TIncludeItemEvent = procedure (const OFN: TOFNotifyEx; var Include: Boolean) of object ; property OnIncludeItem: TIncludeItemEvent;	Возникает для каждого элемента вновь открываемой папки, если свойство Options диалогового окна содержит элемент ofEnableIncludeNotify. Обработчик события с помощью параметра Include указывает, следует ли отображать данный элемент папки
property OnSelectionChange: TNotifyEvent;	Возникает при изменении содержимого диалогового окна: при изменении папки, выбранного файла, фильтра отбора и т. п.
property OnShow: TNotifyEvent;	Возникает при открытии диалогового окна
property OnSelectionChange: TNotifyEvent;	Возникает при изменении фильтра отбора

TOpenPictureDialog и TSavePictureDialog — окна открытия и сохранения изображений

Специализированные диалоговые окна для открытия и сохранения графических файлов TOpenPictureDialog и TSavePictureDialog отличаются от компонентов TOpenDialog и TSaveDialog двумя аспектами. Во-первых, в них предусмотрены стандартные фильтры для выбора графических файлов (с расширениями BMP, ICO, WMF и EMF). Во-вторых, в диалоговые окна включены панели для предварительного просмотра выбираемого файла.

На рис. 19.2 показано стандартное окно компонента TOpenPictureDialog.

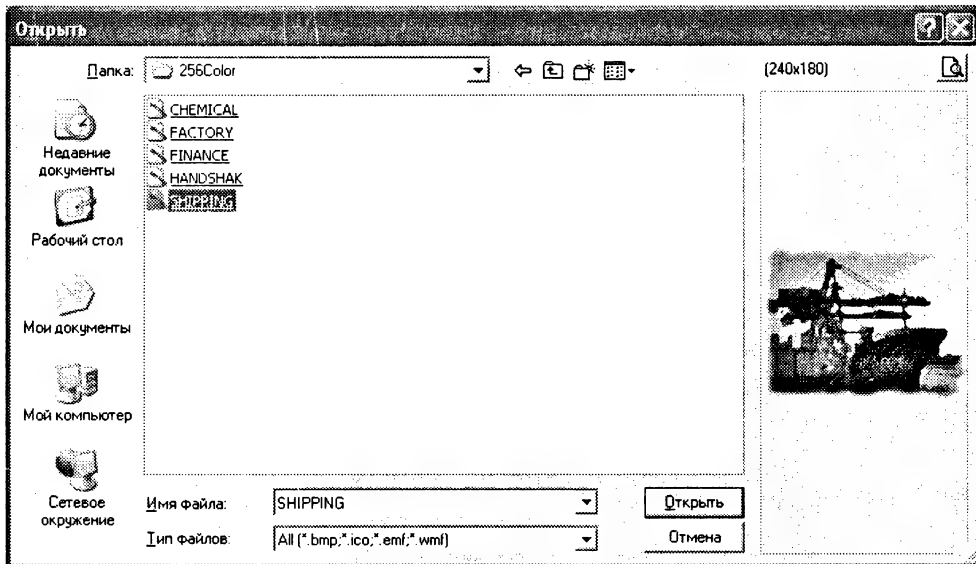


Рис. 19.2. Стандартное окно TOpenPictureDialog

TFontDialog — окно выбора шрифта

Компонент TFontDialog создает и обслуживает стандартное окно выбора шрифта (рис. 19.3).

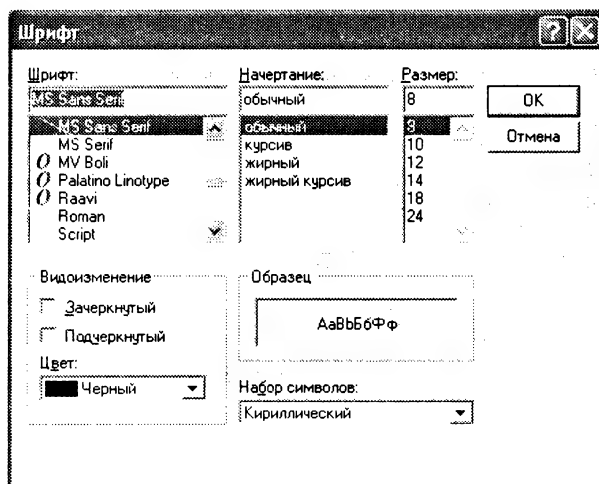


Рис. 19.3. Стандартное окно TFontDialog

Следующее свойство определяет тип устройства, для которого выбирается шрифт:

```
TFontDialogDevice = (fdScreen, fdPrinter, fdBoth);
```

```
property Device: TFontDialogDevice;
```

Здесь: fdScreen — экран; fdPrinter — принтер; fdBoth — шрифты, поддерживаемые и экраном, и принтером.

Диапазон возможных значений размеров шрифтов определяется свойствами MinFontSize и MaxFontSize. Значения этих свойств задаются в *пунктах* (1 пункт равен $1/72$ дюйма, что приблизительно равно 0,36 мм). Если свойства содержат 0, ограничения на размер шрифта отсутствуют.

Представленное ниже свойство используется для настройки диалогового окна:

```
type TFontDialogOption = (fdAnsiOnly, fdTrueTypeOnly,
    fdEffects, fdFixedPitchOnly, fdForceFontExist,
    fdNoFaceSel, fdNoOEMFonts, fdNoSimulations, fdNoSizeSel,
    fdNoStyleSel, fdNoVectorFonts, fdShowHelp, fdWysiwyg,
    fdLimitSize, fdScalableOnly, fdApplyButton);
```

```
TFontDialogOptions = set of TFontDialogOption;
```

```
property Options: TFontDialogOptions;
```

Значения этого свойства имеют следующий смысл:

- fdAnsiOnly — показывает только шрифты с набором символов Windows;
- fdTrueTypeOnly — показывает только TrueType-шрифты;
- fdEffects — включает в окно флажки Подчеркнутый и Зачеркнутый, а также список выбора цвета шрифта;
- ☞ fdFixedPitchOnly — включает только моноширинные шрифты;
- fdForceFontExist — предупреждает о выборе несуществующего шрифта;
- fdNoFaceSel — запрещает выделение имени шрифта в момент открытия окна;
- fdNoOEMFonts — запрещает выбор шрифтов MS-DOS;
- fdNoSimulations — исключает шрифты, которые синтезируются графическим интерфейсом Windows;
- fdNoSizeSel — запрещает выделение размера шрифта в момент открытия окна;
- ☞ fdNoStyleSel — запрещает выделение стиля шрифта в момент открытия окна;
- fdNoVectorFonts — исключает векторные шрифты (шрифты для Windows версии 1.0 используются в плоттерах);
- fdShowHelp — включает в диалоговое окно кнопку Справка;
- fdWysiwyg — включает шрифты, которые поддерживаются и экраном, и принтером;
- fdLimitSize — включает ограничения на размер шрифта, заданные свойствами MaxFontSize и MinFontSize;
- ☞ fdScalableOnly — включает только масштабируемые шрифты (векторные и TrueType-шрифты);
- ☞ fdApplyButton — включает в окно кнопку Применить.

Результат выбора шрифта содержит свойство Font.

TColorDialog — окно выбора цвета

Компонент создает и обслуживает стандартное диалоговое окно выбора цвета, показанное на рис. 19.4.

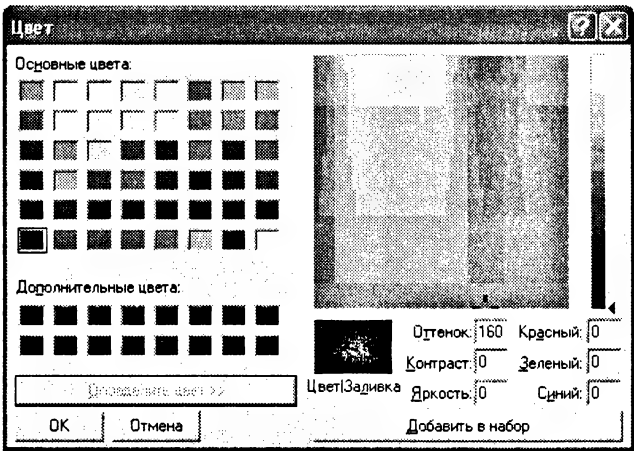


Рис. 19.4. Стандартное окно компонента TColorDialog

Свойства компонента TColorDialog представлены в табл. 19.2.

Таблица 19.2. Свойства компонента TColorDialog

Свойство	Описание
property Color: TColor;	Содержит выбранный цвет
property CustomColors: TStrings;	Содержит до 16 цветов, определенных пользователем. Каждая строка имеет формат ColorX=nnnnnn, где X — буква от A до P, определяющая номер цвета, nnnnnn — шестнадцатеричное представление цвета в модели RGB (красный, зеленый, голубой)
TColorDialogOption = (cdFullOpen, cdPreventFullOpen, cdShowHelp, cdSolidColor, cdAnyColor); TColorDialogOptions = set of TColorDialogOption; property Options: TColorDialogOptions;	Определяет вид окна: cdFullOpen — показывать с развернутой палитрой выбора цвета; cdPreventFullOpen — запретить показ палитры выбора цвета; cdShowHelp — включить в окно кнопку Справка; cdSolidColor — выбирать ближайший сплошной цвет; cdAnyColor — разрешить выбор несплошных цветов

TPrintDialog — окно настройки параметров печати

Компонент TPrintDialog создает стандартное диалоговое окно для выбора параметров печати, показанное на рис. 19.5.

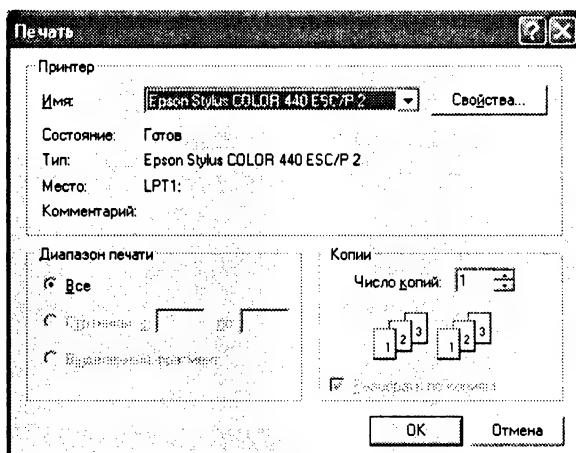


Рис. 19.5. Стандартное окно компонента TPrintDialog.

Свойства компонента TPrintDialog перечислены в табл. 19.3.

Таблица 19.3. Свойства компонента TPrintDialog

Свойство	Описание
property Collate: Boolean;	Если имеет значение True, окно отображается с установленным флажком Разобрать по копиям (Collate). Если этот флажок установлен, печать нескольких копий документа будет идти по копиям: сначала первая копия, затем вторая и т. д., в противном случае — по страницам: сначала все копии первой страницы, затем второй и т. д.
property Copies: Integer;	Определяет количество копий (0 — одна копия)
property FromPage: Integer;	Определяет начальную страницу печати
property MaxPage: Integer;	Определяет верхнюю границу диапазона страниц для свойств FromPage и ToPage
property MinPage: Integer;	Определяет нижнюю границу диапазона страниц для свойств FromPage и ToPage
TPrintDialogOption = (poPrintToFile, poPageNums, poSelection, poWarning, poHelp, poDisablePrintToFile); TPrintDialogOptions = set of TPrintDialogOption; property Options: TPrintDialogOptions;	Определяет настройку окна: poPrintToFile — печатать в файл; poPrintToFile — разрешает выбор диапазона страниц; poSelection — разрешает печать выбранного текста; poWarning — предупреждать пользователя о не установленном принтере; poHelp — вставить в окно кнопку Справка; poDisablePrintToFile — запрещает печать в файл
TPrintRange = (prAllPages, prSelection, prPageNums); property PrintRange: TPrintRange;	Определяет диапазон печатаемых страниц: prAllPages — все страницы; prSelection — выделенный фрагмент текста; prPageNums — страницы по номерам

продолжение ➤

Таблица 19.3 (продолжение)

Свойство	Описание
property PrintToFile: Boolean;	Содержит значение True, если пользователь выбрал печать в файл
property ToPage: Integer;	Определяет конечную страницу печати

TPrinterSetupDialog — окно настройки параметров принтера

Компонент TPrinterSetupDialog создает диалоговое окно настройки параметров принтера, вид которого зависит от типа принтера. Это окно взаимодействует с драйвером принтера и не возвращает в программу никакой информации, поэтому его метод Execute — процедура, а не функция.

TFindDialog — окно поиска

Стандартное диалоговое окно компонента TFindDialog используется для поиска фрагмента текста (рис. 19.6).

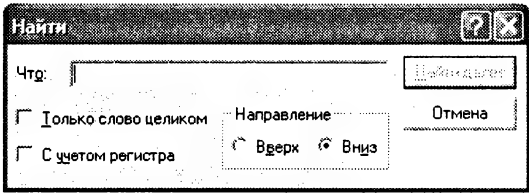


Рис. 19.6. Стандартное окно компонента TFindDialog.

Свойства компонента TFindDialog перечислены в табл. 19.4.

Таблица 19.4. Свойства компонента TFindDialog

Свойство	Описание
property FindText: String;	Определяет образец поиска
property Left: Integer;	Содержит горизонтальную позицию левого верхнего угла места появления окна
property Options: TFindOptions;	Определяет настройку диалогового окна (см. ниже)
property Position: TPoint;	Содержит горизонтальную и вертикальную позиции левого верхнего угла места появления окна
property Top: Integer;	Содержит вертикальную позицию левого верхнего угла места появления окна

Для компонента определен следующий тип, использующийся в свойстве Options:

```

type TFindOption = (frDown, frFindNext, frHideMatchCase,
                    frHideWholeWord, frHideUpDown,
                    frMatchCase, frDisableMatchCase,
                    frDisableUpDown, frDisableWholeWord,
                    frReplace, frReplaceAll, frWholeWord,
                    frShowHelp);

```

```

TFindOptions = set of TFindOption;

```

Его значения имеют следующий смысл:

- frDown — устанавливает поиск вперед по тексту;
- frFindNext — сообщает программе, что пользователь щелкнул на кнопке Найти далее;
- frHideMatchCase — снимает флажок С учетом регистра;
- frHideWholeWord — снимает флажок Только слово целиком;
- frHideUpDown — скрывает переключатели выбора направления поиска;
- frMatchCase — устанавливает флажок С учетом регистра;
- frDisableMatchCase — делает недоступным флажок С учетом регистра;
- frDisableUpDown — запрещает выбор направления поиска;
- frDisableWholeWord — делает недоступным флажок Только слово целиком;
- frReplace — используется в компоненте TReplaceDialog и указывает на необходимость замены найденного фрагмента;
- frReplaceAll — используется в компоненте TReplaceDialog и указывает на необходимость замены всех вхождений образца поиска;
- frWholeWord — устанавливает флажок Только слово целиком;
- frShowHelp — включает в окно кнопку Справка.

Специфичной для диалогового окна является реализуемая им возможность просмотра найденного фрагмента и при необходимости продолжение поиска. С этой целью для компонента определено событие OnFind: TNotifyEvent, которое возникает всякий раз, когда пользователь щелкает на кнопке Найти далее. Обработчик события должен найти образец в тексте и показать его пользователю. Пусть, например, компонент Memo1 содержит отыскиваемый фрагмент, и поиск идет с учетом регистра. Тогда обработчик может иметь такой вид:

```

procedure TForm1.FindDialog1Find(Sender: TObject);
var
    SelPos: Integer;
begin
    with TFindDialog(Sender) do
        begin
            // Ищем фрагмент в тексте
            SelPos := Pos(FindText, Memo1.Lines.Text);
            if SelPos > 0 then

```

```

begin // Фрагмент найден - выделяем его
  Mem1.SelStart := SelPos - 1;
  Mem1.SelLength := Length(FindText);
end else
  ShowMessage('Текст "' + FindText + '" не найден')
end;
end;
end;

```

TReplaceDialog — окно поиска и замены

Компонент TReplaceDialog создает и обслуживает окно поиска и замены текстового фрагмента (рис. 19.7).

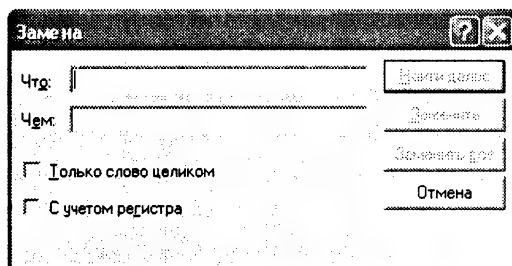


Рис. 19.7. Стандартное окно компонента TReplaceDialog

Класс TReplaceDialog является прямым потомком класса TFindDialog и наследует от него большую часть свойств. Дополнительно в компоненте определено

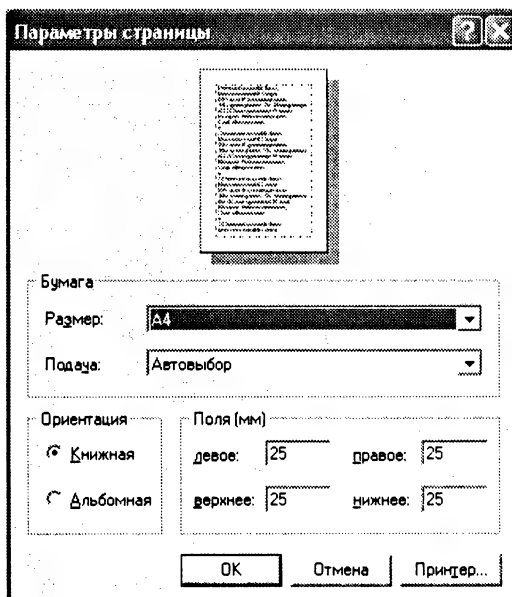


Рис. 19.8. Стандартное окно компонента TPageSetupDialog

свойство `ReplaceText`: **String**, в котором содержится текст замены, и событие `OnReplace`, которое возникает при щелчке на кнопке **Заменить** или **Заменить все**.

TPageSetupDialog — окно установки параметров печатаемой страницы

Компонент `TPageSetupDialog` создает и обслуживает окно настройки параметров печатаемой страницы (рис. 19.8).

Свойства и методы компонента предоставляют пользователю гибкие средства управления параметрами печатаемой страницы (ее ориентацией, размерами полей и т. п.). С помощью обработчиков событий программист может предоставить пользователю возможность изменения указанных параметров в различных ситуациях — перед началом печати, перед печатью очередной страницы и т. д. Подробнее о компоненте см. в справочной службе Delphi.

Глава 20

Компоненты вкладки Samples

Компоненты вкладки Samples включены в Delphi как образцы (samples) разработки нестандартных компонентов, однако многие из них имеют вполне самостоятельное значение.

TGauge — индикатор величины

Компонент TGauge предназначен для отображения некоторой изменяющейся числовой величины. Он отличается от компонента TProgressBar разнообразием форм (рис. 20.1).

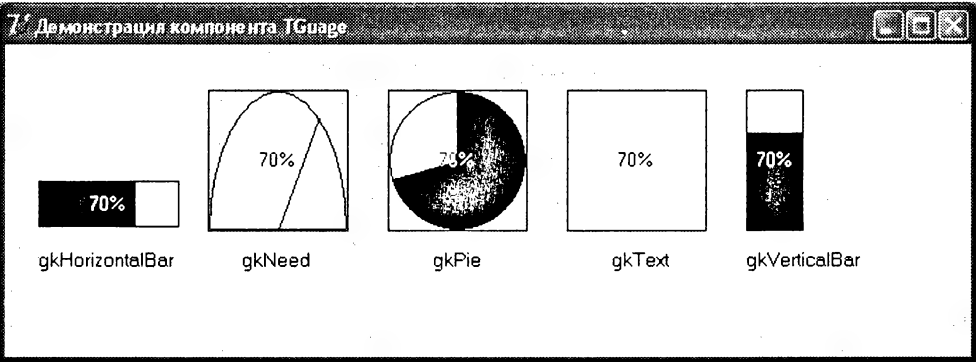


Рис. 20.1. Различные формы компонента TGauge

Свойства компонента TGauge перечислены в табл. 20.1.

Таблица 20.1. Свойства компонента TGauge

Свойство	Описание
property BackColor: TColor;	Цвет незакрашенной части индикатора
property ForeColor: TColor;	Цвет закрашенной части индикатора

Свойство	Описание
<code>TGaugeKind = (gkText, gkHorizontalBar, gkVerticalBar, gkPie, gkNeedle);</code> property <code>Kind: TGaugeKind;</code>	Определяет форму индикатора (см. рис. 20.1)
property <code>MaxValue: LongInt;</code>	Определяет максимальное значение диапазона изменения свойства <code>Progress</code>
property <code>MinValue: LongInt;</code>	Определяет минимальное значение диапазона изменения свойства <code>Progress</code>
property <code>PercentDone: LongInt;</code>	Содержит текущее значение свойства <code>Progress</code> в процентах от его диапазона изменения
property <code>Progress: LongInt;</code>	Содержит текущее значение изменяющейся числовой величины
property <code>ShowText: Boolean;</code>	Если содержит значение <code>True</code> , в центре компонента выводится строковое представление значения <code>PercentDone</code>

С помощью следующего метода к текущему значению свойства `Progress` можно добавить величину `Value`:

```
procedure AddProgress(Value: LongInt);
```

TColorGrid — цветовая таблица

Компонент `TColorGrid` предназначен для выбора и/или отображения цветов из 16-цветной палитры. Доступны два цвета: основной и фоновый. Основной цвет выбирается щелчком левой кнопки мыши и отображается символами `FG`, фоновый выбирается правой кнопкой мыши и отображается символами `BG` (если оба цвета совпадают, соответствующая ячейка таблицы помечается символами `FB`).

Свойства компонента `TColorGrid` представлены в табл. 20.2.

Таблица 20.2. Свойства компонента `TColorGrid`

Свойство	Описание
property <code>BackgroundColor: TColor;</code>	Содержит выбранный фоновый цвет
property <code>BackgroundEnabled: Boolean;</code>	Разрешает/запрещает выбор фонового цвета. Игнорируется, если свойство <code>ClickEnablesColor</code> имеет значение <code>True</code>
property <code>BackgroundIndex: Integer;</code>	Содержит индекс выбранного фонового цвета
property <code>ClickEnablesColor: Boolean;</code>	Разрешает/запрещает выбор цветов
type <code>TGridOrdering = (go16x1, go8x2, go4x4, go2x8, go1x16);</code>	Определяет конфигурацию таблицы: первая цифра — количество колонок.

продолжение ➤

Таблица 20.2 (продолжение)

Свойство	Описание
property GridOrdering: TGridOrdering;	вторая — количество рядов
property ForegroundColor: TColor;	Содержит выбранный основной цвет
property ForegroundEnabled: Boolean;	Разрешает/запрещает выбор основного цвета. Игнорируется, если свойство ClickEnablesColor имеет значение True
property ForegroundIndex: Integer;	Содержит индекс выбранного основного цвета
property Selection: Integer;	Содержит индекс последнего выбранного цвета

Индекс цвета — это число в диапазоне 0–15, определяющее номер соответствующей ячейки в порядке перехода от самых темных цветов к самым светлым: 0 — ячейка с черным, 1 — с коричневым, ... , 15 — с белым цветом.

Пары свойств BackgroundEnabled-ClickEnablesColor и ForegroundEnabled-ClickEnablesColor запрещают выбор и отображение соответствующего цвета только в том случае, когда *оба* свойства в паре имеют значение False. Однако даже в этом случае свойство Selection хранит номер ячейки таблицы, на которой последний раз был произведен щелчок левой или правой кнопкой мыши. Это свойство изменяется до возникновения события мыши и при необходимости может использоваться для выбора цвета без его отображения.

С помощью представленного ниже метода программа может получить индекс табличного цвета, наиболее близкого к цвету AColor:

```
function ColorToIndex(AColor: TColor): Integer;
```

Следующее событие возникает только при разрешенном выборе/отображении соответствующего цвета:

```
property OnChange: TNotifyEvent;
```

TSpinButton — спаренные кнопки

В отличие от компонента TUpDown, компонент TSpinButton не связан с регулированием числовой величины и может использоваться просто как пара кнопок: верхняя (Up) и нижняя (Down). Для компонента следует написать обработчики его событий OnUpClick и OnDownClick, в которых реализуется реакция программы на щелчок на той или иной кнопке.

Рисунки на кнопках определяются свойствами UpGlyph и DownGlyph. Каждый рисунок может содержать 1, 2 или 3 изображения, соответственно, для кнопки в обычном состоянии, для недоступной кнопки и для нажатой кнопки.

TSpinEdit — поле со спаренными кнопками

Компонент TSpinEdit предназначен для ввода/отображения числа. Он представляет собой комбинацию специализированного потомка однострочного текстового

поля TCustomEdit и кнопок TSpinButton. В текстовое поле допустимо вводить только числа (в этом и заключается его специализация), а кнопки используются для изменения числа на некоторую фиксированную величину.

Свойства компонента TSpinEdit перечислены в табл. 20.3.

Таблица 20.3. Свойства компонента TSpinEdit

Свойство	Описание
property EditorEnabled: Boolean;	Разрешает/запрещает ручной ввод чисел в текстовое поле
property Increment: LongInt;	Содержит число, на которое будет изменяться свойство Value при щелчке на одной из кнопок TSpinButton
property MaxValue: LongInt;	Содержит верхнюю границу диапазона возможных значений Value
property MinValue: LongInt;	Содержит нижнюю границу диапазона возможных значений Value
property Value: LongInt;	Содержит текущее значение числовой величины

TDirectoryOutLine — структура каталогов

Компонент TDirectoryOutLine предназначен для отображения древовидной структуры каталогов (рис. 20.2). Компонент отображает полную структуру каталогов, а не путь доступа к одному из них.

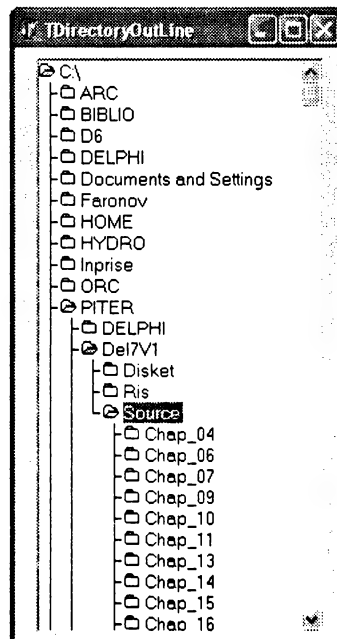


Рис. 20.2. Окно компонента TDirectoryOutLine

Следующее свойство определяет регистр букв в именах каталогов:

```
TTextCase = (tcLowerCase, tcUpperCase, tcAsIs);  
property TextCase: TTextCase;
```

Здесь: tcLowerCase — строчные буквы; tcUpperCase — прописные буквы; tcAsIs — строчные и прописные буквы.

Следующие два свойства доступны только на этапе прогона программы:

```
property Drive: Char;  
property Directory: TFileName;
```

Первое определяет выбранный диск, второе — выбранный каталог.

TCalendar — календарь

Компонент TCalendar отображает календарь на выбранный месяц и год. Его свойства Day, Month и Year могут содержать любую дату от 1 до 9999 года от Рождества Христова.

Свойства компонента TCalendar перечислены в табл. 20.4.

Таблица 20.4. Свойства компонента TCalendar

Свойство	Описание
property CalendarDate: TDateTime;	Содержит выбранную дату
property CellText[ACol, ARow: Integer]: String;	Содержит текст в ячейке календаря на пересечении колонки ACol с рядом ARow (только для чтения)
property Day: Integer;	Определяет выбранный день месяца
property GridLineWidth: Integer;	Устанавливает толщину линий таблицы календаря
property Month: Integer;	Определяет выбранный месяц
property ReadOnly: Boolean;	Если содержит значение True, пользователь не может выбрать другую дату
type TDayOfWeek = 0..6; property StartOfWeek: TDayOfWeek;	Определяет день недели, который будет отображаться в левой колонке таблицы
property UseCurrentDate: Boolean;	Если содержит значение True, компонент будет показывать текущую системную дату
property Year: Integer;	Определяет выбранный год

Методы компонента TCalendar представлены в табл. 20.5.

Таблица 20.5. Методы компонента TCalendar

Метод	Описание
procedure NextMonth;	Отображает календарь на следующий месяц
procedure NextYear;	Отображает календарь на следующий год

Метод	Описание
procedure PrevMonth;	Отображает календарь на предыдущий месяц
procedure PrevYear;	Отображает календарь на предыдущий год

Компоненты Windows Shell

Компоненты TShellComboBox, TShellListView, TShellTreeView и TShellChangeNotifier предназначены для более полного взаимодействия с файловой системой 32-разрядных версий Windows. Окно, показанное на рис. 20.3 (проект Chap_20\WinShell\WinExplorer.dpr) в функциональном плане очень напоминает окно Проводника Windows.

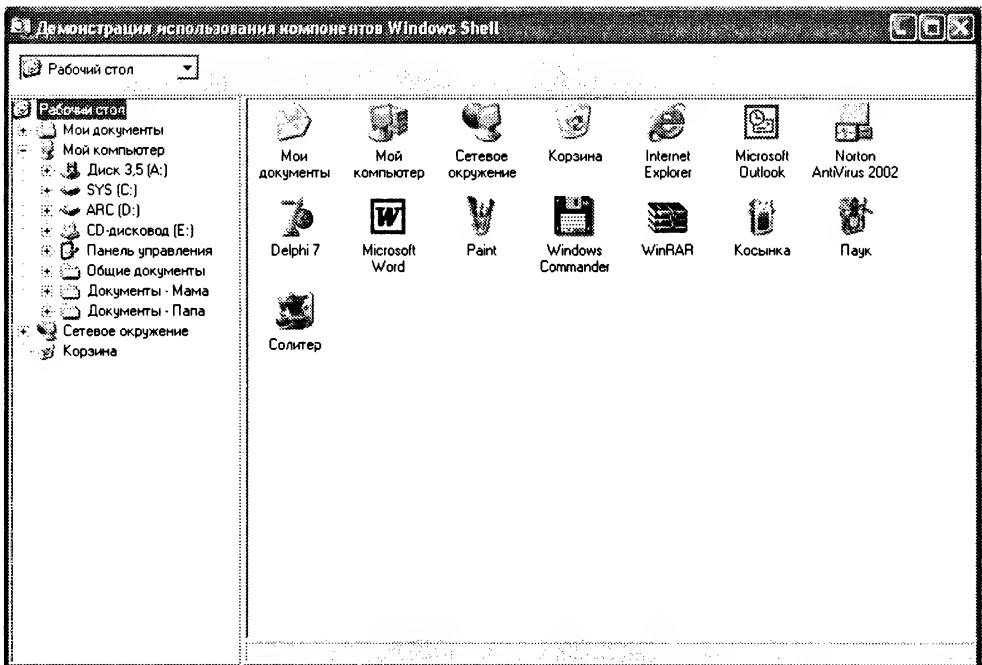


Рис. 20.3. Демонстрация компонентов Windows Shell

Для повторения примера проделайте следующую последовательность шагов.

1. На пустую форму поместите две панели. Для первой установите в свойство Align значение alTop, для второй — alLeft.
2. Поместите на форму компонент TSplitter, установите для него значения свойств Align=alLeft и Width=3.
3. Поместите на форму еще одну панель и установите для нее значение свойства Align=alClient.
4. На верхнюю панель поместите компонент TShellComboBox, на правую — компонент TShellTreeView, на левую — компоненты TShellListView и TSta-

tusBar. Поместите в свойства Align компонентов TShellTreeView и TShellListView значения alClient.

5. Свяжите три компонента друг с другом: в свойство ShellTreeView компонента TShellComboBox поместите ссылку на компонент ShellTreeView1, а в его свойство ShellListView — ссылку на компонент ShellListView1. Убедитесь, что свойства ShellComboBox и ShellListView компонента ShellTreeView1, а также свойства ShellComboBox и ShellTreeView компонента ShellListView1 настроены нужным образом.
6. Чтобы в строке состояния отображать название просматриваемого устройства (папки), напишите два показанных ниже обработчика:

```

procedure TForm1.ShellTreeView1Change(Sender: TObject;
                                         Node: TTreeNode);
begin
    if Tag=1 then
        Panel.Text := Node.Text
end;

procedure TForm1.FormActivate(Sender: TObject);
begin
    Panel := StatusBar1.Panels.Add;
    Tag := 1
end;

```

Глава 21

Форма

Форма является основным «строительным блоком» в Delphi. Любая программа имеет как минимум одну связанную с ней форму, которая называется главной, — эта форма появляется на экране в момент старта программы. Однако программа может иметь сколько угодно форм, каждая из которых может решать ту или иную локальную задачу и появляться на экране по мере надобности. В этой главе мы познакомимся с назначением и способами использования различных форм, а также изучим их свойства и методы.

Разновидности форм

Разновидности форм определяются значениями их свойства `FormStyle`, а также разнообразием форм-заготовок в хранилище объектов Delphi.

Стиль формы задается одним из значений следующего свойства:

```
TFormStyle = (fsNormal, fsMDIChild, fsMDIForm,  
              fsStayOnTop);
```

```
property FormStyle: TFormStyle;
```

Стиль `fsNormal` определяет обычную форму, использующуюся для решения самых разных задач, в том числе для общего управления всей программой (главная форма).

Стили `fsMDIChild` и `fsMDIForm` используются при создании так называемых многодокументных приложений в стиле MDI (Multi Document Interface). Этот немодный сегодня стиль предполагает создание главного окна MDI (его обычно называют рамочным), внутри которого по мере надобности появляются дочерние окна. Дочерние окна, подобно дочерним элементам контейнера, не могут выходить за границы своего владельца — рамочного окна. В MDI-приложениях есть специальные средства управления взаимодействием рамочного окна с дочерними окнами. Например, каждое дочернее окно в момент активизации может нужным образом настроить главное меню рамочного окна (дочерние MDI-окна не имеют собственного главного меню). В Delphi для создания рамочного окна используется стиль `fsMDIForm`, а для создания дочернего MDI-окна — стиль `fsMDIChild`.

Стиль `fsStayOnTop` предназначен для окон, которые всегда должны располагаться над всеми другими окнами программы¹. В момент активизации окна оно обычно становится видимым на экране, даже если перед этим его загораживали другие раскрытые окна. Стиль `fsStayOnTop` препятствует перекрытию окна другими окнами, даже если оно становится неактивным и теряет фокус ввода (так сказано в документации, однако на самом деле это не так, см. ниже). Понятно, что этот стиль используется в исключительных случаях, когда окно содержит что-то, требующее повышенного внимания пользователя.

Как показывает практика, объявление окна со стилевым признаком `fsStayOnTop` еще не решает проблему создания всплывающего окна, то есть окна, которое невозможно перекрыть другими окнами. Более того, несложные эксперименты убедят вас, что этот признак вообще не играет никакой роли! Чтобы создать всплывающее окно, нужно обратиться к API-функции `SetWindowPos`. Например:

```
SetWindowPos(fmAlarmForm.Handle, hwnd_TopMost,
             300, 300, 250, 70, swp_noActivate)
```

Полное описание этой функции (как и всех других API-функций Windows) вы найдете в файле помощи `WIN32.HLP` или `WIN32SDK.HLP`, которые поставляются вместе с Delphi и по умолчанию располагаются в папке `Program Files\Common Files\Borland Shared\MSHlp`. Здесь я лишь кратко поясню назначение параметров вызова.

Первым параметром обращения к функции является дескриптор окна, которое должно стать всплывающим. Напомню, что дескриптор формы содержится в ее свойстве `Handle`, поэтому для формы `fmAlarmForm` параметр вызова имеет вид `fmAlarmForm.Handle`. Второй параметр определяет расположение окна относительно других окон в так называемом Z-порядке их расположения. Константа `hwnd_TopMost` указывает, что окно должно стать самым верхним и впредь до его закрытия не может перекрываться другими (обычными) окнами. Четыре следующих параметра определяют координаты левого верхнего угла окна, его ширину и высоту. Все параметры указываются в пикселах, координаты угла задаются относительно левого верхнего угла экрана. Последним указывается один или несколько битовых флагов, уточняющих поведение окна. В нашем примере использован флаг `swp_noActivate`, означающий, что окно не получает фокуса ввода в момент своего появления на экране. На практике я часто использую всплывающие окна, чтобы сообщить пользователю, например, о необходимости обновить набор данных, которые он видит в настоящий момент, так как эти данные были изменены другими пользователями клиент-серверной базы данных. Такое окно, в отличие от модального окна, не должно отнимать активность (фокус ввода) у окна, с которым работает пользователь.

И последнее замечание. Всплывающим окном может стать и обычное окно (со значением стиля `FormStyle=fsNormal`). Однако всплывающее окно действительно появится на экране, только если в его свойстве `Visible` содержится значение `True`. Иными словами, если пользователь закроет такое окно, оно не появится в результате вызова функции `SetWindowPos` — его предварительно нужно сделать видимым методом `Show`. Поскольку всплывающее окно может вызываться в раз-

¹ В терминологии Windows такое окно называется «*Popup Window*», что буквально переводится как «окно-поплавок», а в локализованных программных продуктах именуется «всплывающим окном».

ных местах программы, удобно поместить вызов функции `SetWindowsPos` в его обработчик события `OnActivate`. Однако если вы попытаетесь создать такой обработчик для окна со стилем `FormStyle=fsStayOnTop`, то компилятор выдаст сообщение об ошибке и поместит заготовку метода в тексте модуля после ограничителя-точки (`.`). В этом случае определяйте окно со стилем `fsNormal`.

Современные многооконные приложения чаще всего строятся в стиле SDI (Single Document Interface), который в противоположность интерфейсу MDI не накладывает ограничений на положение и размеры вспомогательных форм, каждая из которых при необходимости может иметь свое главное меню (в стиле SDI реализована, например, среда Delphi). Для создания форм в этом случае используется стиль `fsNormal`. Замечу, что в рамках SDI-приложений могут использоваться рамочные MDI-формы со своими дочерними окнами, так что термин SDI носит весьма условный характер и применяется в основном для противопоставления термину MDI.

В хранилище объектов Delphi хранится множество стандартных форм-заготовок, предназначенных для решения конкретных задач (доступ к хранилищу объектов открывает команда `File ► New ► Other`). Помимо универсальной пустой формы `Form` (вкладка `New` хранилища объектов) хранилище содержит специализированные формы, некоторые из которых перечислены в табл. 21.1.

Таблица 21.1. Специализированные формы в хранилище объектов

Название	Вкладка	Описание
About box	Forms	Окно 0 программе
Dual list box	Forms	Диалоговое окно с двумя компонентами <code>Listbox</code> . Используется для гибкого управления списками, в том числе для перемещения элементов из одного списка в другой
Quick Report Labels	Forms	Используется в приложениях баз данных для печати наклеек
Quick Report List	Forms	Используется в приложениях баз данных для создания обычных отчетов
Quick Report Master/Detail	Forms	Используется в приложениях баз данных для создания отчетов типа главный-детальный
Tabbed Pages	Forms	Заготовка для многостраничного диалогового окна с вкладками и кнопками <code>OK</code> , <code>Cancel</code> и <code>Help</code>
Dialog with Help	Dialogs	Заготовка для диалогового окна с кнопками <code>OK</code> , <code>Cancel</code> , <code>Help</code> . Имеются варианты с вертикальным и горизонтальным расположением кнопок
Password Dialog	Dialogs	Диалоговое окно с однострочным полем <code>TEdit</code> для ввода паролей, а также кнопками <code>OK</code> и <code>Cancel</code>
Reconcile Error Dialog	Dialogs	Используется в приложениях баз данных для пояснения обнаруженной ошибки при изменении таблицы
Standard Dialog	Dialogs	Заготовка для диалогового окна с кнопками <code>OK</code> , <code>Cancel</code> . Имеются варианты с вертикальным и горизонтальным расположением кнопок

— продолжение ➤

Таблица 21.1 (продолжение)

Название	Вкладка	Описание
Dialog Wizard	Dialogs	Мастер создания диалоговых окон
Database Form Wizard	Business	Мастер создания форм для доступа к базам данных
Quick Report Wizard	Business	Мастер создания отчетов для баз данных
TeeChart Wizard	Business	Мастер форм для доступа к компоненту Chart

Компонент TForm

В этом разделе мы рассмотрим свойства, методы и события формы, то есть компонента TForm. Свойства формы представлены в табл. 21.2.

Таблица 21.2. Свойства формы

Свойство	Описание
property Active: Boolean;	Содержит значение True, если окно активно (имеет фокус ввода)
property ActiveControl: TWinControl;	Определяет дочерний элемент, содержащий фокус ввода
property ActiveMDIChild: TForm;	Определяет дочернее MDI-окно с фокусом ввода
property AlphaBlend: Boolean;	Если содержит значение True, форма определяет полупрозрачное окно (только для Windows 2000 и выше)
property AlphaBlendValue: Byte;	Указывает степень прозрачности окна
TBorderIcon = (biSystemMenu, biMinimize, biMaximize, biHelp); TBorderIcons = set of TBorderIcon; property BorderIcons: TBorderIcons;	Определяет наличие кнопок в заголовке окна: biSystemMenu — кнопка вызова системного меню; biMinimize — кнопка свертывания; biMaximize — кнопка разворачивания; biHelp — кнопка вызова справочной службы
TFormBorderStyle = (bsNone, bsSingle, bsSizeable, bsDialog, bsToolWindow, bsSizeToolWin); property BorderStyle: TFormBorderStyle;	Определяет стиль рамки окна: bsNone — окно не имеет рамки и заголовка и не может перемещаться и изменять свои размеры; bsSingle — рамка толщиной в 1 пиксел, такое окно не может изменять свои размеры; bsSizeable — обычная рамка; bsDialog — рамка диалогового окна, окно не может изменять свои размеры; bsToolWindow — подобно bsSingle, но с уменьшенным по высоте заголовком; bsSizeToolWin — подобно bsSizeable, но с уменьшенным по высоте заголовком

Свойство	Описание
property Canvas: TCanvas;	Канва для прорисовки фона окна. Это свойство могут использовать неоконные дочерние элементы
property ClientHeight: Integer;	Высота клиентской части окна
property ClientRect: TRect;	Прямоугольник клиентской части окна
property ClientWidth: Integer;	Ширина клиентской части окна
type TDefaultMonitor = (dmDesktop, dmPrimary, dmMainForm, dmActiveForm); property DefaultMonitor: TDefaultMonitor;	Определяет монитор, на котором появляется форма: dmDesktop — для формы не указан монитор; dmPrimary — форма появляется на первом мониторе списка Monitors глобального объекта Screen; dmMainForm — форма появляется на том же мониторе, что и главное окно программы; dmActiveForm — форма появляется на том же мониторе, что и текущее активное окно
property DropTarget: Boolean;	Если содержит значение True, форма поддерживает перетаскивание (Drag&Drop)
property Floating: Boolean;	Если содержит значение False, окно причалено к другому окну, в противном случае оно «плавает»
type TFormState = set of (fsCreating, fsVisible, fsShowing, fsModal, fsCreatedMDIChild, fsActivated); property FormState: TFormState;	Определяет текущее состояние формы: fsCreating — окно создается; fsVisible — окно видно на экране (это состояние используется для обновления свойства Visible); fsShowing — свойство FormState изменяется; fsModal — создано модальное окно; fsCreatedMDIChild — форма представляет собой рамочное MDI-окно (устанавливается в момент создания дочернего окна); fsActivated — форма получила сообщение CM_ACTIVATE, но еще не возникло событие OnActivate
type TFormStyle = (fsNormal, fsMDIChild, fsMDIForm, fsStayOnTop); property FormStyle: TFormStyle;	Определяет стиль окна: fsNormal — обычное окно; fsMDIChild дочернее MDI-окно; fsMDIForm — рамочное MDI-окно; fsStayOnTop — всплывающее окно
property HelpFile: String;	Каждая форма может иметь индивидуальный файл помощи, имя которого содержит это свойство. Если имя не указано, используется файл помощи приложения

Таблица 21.2 (продолжение)

Свойство	Описание
property Icon: TIcon;	Содержит значок окна. Для главной формы это свойство определяет также значок программы
property KeyPreview: Boolean;	Если имеет значение True, форма получает события от клавиатуры, перед тем как они поступят в элемент с фокусом ввода
property MDIChildCount: Integer;	В рамочном MDI-окне определяет количество связанных с ним дочерних MDI-окон
property MDIChildren[N: Integer]: TForm;	В рамочном MDI-окне открывает доступ к дочернему окну под номером N
property Menu: TMainMenu;	Содержит главное меню окна
TModalResult = Low(Integer).. High(Integer); property ModalResult: TModalResult;	Для модального окна содержит результат диалога с пользователем
property Monitor: TMonitor;	Содержит ссылку на монитор, на котором отображается окно
property PixelsPerInch: Integer;	Определяет разрешающую способность окна в пикселах на один линейный дюйм для этапа конструирования формы
TPosition = (poDesigned, poDefault, poDefaultPosOnly, poDefaultSizeOnly, poScreenCenter); property Position: TPosition;	Определяет положение и размеры окна в момент его появления на экране: poDesigned — такие же, как на этапе конструирования окна; poDefault — положение и размеры определяет Windows; poDefaultPosOnly — положение, как на этапе конструирования, размеры определяет Windows; poDefaultSizeOnly — размеры, как на этапе конструирования, положение определяет Windows; poScreenCenter — в центре экрана с размерами, как на этапе конструирования
TPrintScale = (poNone, poProportional, poPrintToFit); property PrintScale: TPrintScale;	Определяет масштабирование окна при его печати на принтере: poNone — нет масштабирования; каждый пиксел окна воспроизводится одной точкой на бумаге; poProportional — форма масштабируется так, чтобы ее образ на бумаге был максимально похож на ее изображение на экране; poPrintToFit — форма печатается с такими же

Свойство	Описание
	пропорциями, как на экране, но с размерами, заполняющими лист бумаги
property Scaled: Boolean;	Разрешает/запрещает масштабировать форму, если значение ее свойства PixelPerInch отличается от текущего разрешения экрана
TileMode = (tbHorizontal, tbVertical); property TileMode: TTileMode;	Определяет стиль расположения дочерних окон MDI-приложения при их упорядочении мозаикой
property TransparentColor: Boolean;	Если содержит значение True, свойство TransparentColorValue определяет цвет прозрачности формы, сквозь который будут видны нижележащие окна (только для Windows 2000 и выше)
property TransparentColorValue: TColor;	Содержит цвет прозрачности формы (игнорируется, если TransparentColor=False)
property WindowMenu: TMenuItem;	Определяет пункт главного меню рамочного MDI-окна, к которому добавляются пункты меню дочернего окна
TWindowState = (wsNormal, wsMinimized, wsMaximized); property WindowState: TWindowState;	Определяет состояние окна в момент его появления на экране: wsNormal — обычное окно; wsMinimized — свернуто до кнопки; wsMaximized — развернуто на весь экран

Два свойства AlphaBlend и AlphaBlendValue впервые введены в версии 6 и позволяют регулировать степень прозрачности окна формы. Если AlphaBlend=True, то значение свойства AlphaBlendValue задает степень прозрачности: 0 — окно полностью прозрачно, 255 — окно совершенно непрозрачно. Свойства TransparentColorValue и TransparentColor определяют цвет прозрачности и возможность его использования. Однако указанные свойства работают только под управлением Windows 2000/XP и на процессорах Pentium с тактовой частотой не ниже 90 МГц.

Замечу, что результат изменения значения AlphaBlendValue проявляется только в том случае, если видеокарта компьютера способна отображать больше 256 цветов, то есть использует два (High Color) или более (True Color) байта для кодирования цвета одного пиксела. Свойства TransparentColorValue и TransparentColor игнорируются, если цвет не относится к основной 16-цветной палитре или если свойство AlphaBlend содержит значение True. На рис. 21.1 показан экран, созданный с помощью проекта Chap_21\Transparent Window\Transp.dpr.

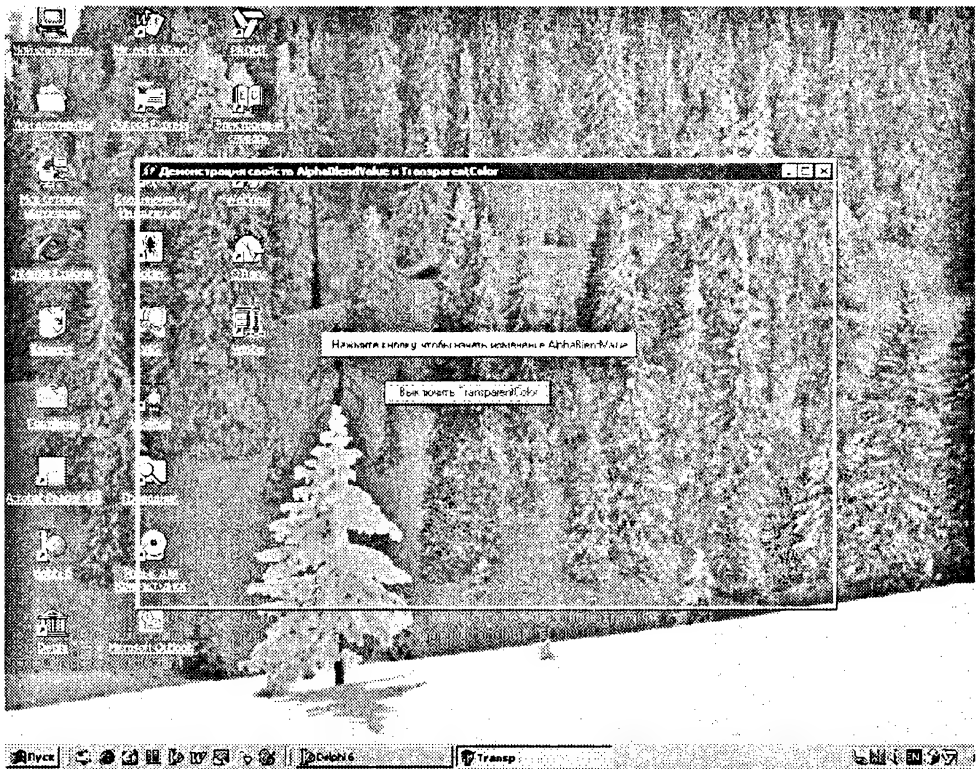


Рис. 21.1. Демонстрация свойств AlphaBlendValue и TransparentColor

Показываемое окно действительно прозрачно — сквозь него видны не только значки рабочего стола, но на них даже можно щелкать мышью и вызывать программы. После щелчка на верхней кнопке начинается плавное изменение значения AlphaBlendValue от 255 до 0 и обратно. В результате окно «растворяется» на экране, а затем появляется вновь.

ПРИМЕЧАНИЕ Если в множестве свойства BorderIcon убрать кнопки biMinimize и biMaximize, а в свойство WindowState поместить значение wsMaximized, форма займет все пространство экрана, включая панель задач.

Методы формы перечислены в табл. 21.3.

Таблица 21.3. Методы формы

Метод	Описание
procedure ArrangeIcons;	Упорядочивает значки закрытых дочерних окон MDI-приложения
procedure Cascade;	Располагает дочерние MDI-окна каскадом
procedure Close;	Закрывает окно. Для главного окна завершает работу программы

Метод	Описание
function CloseQuery: Boolean	Возвращает значение True, если можно закрыть окно
procedure DefocusControl (Control: TWinControl; Removing: Boolean);	Отбирает фокус ввода у дочернего элемента Control. Если при этом Removing=True, фокус ввода получает форма
procedure CreateParams (var Params: TCreateParams); override;	Создает структуру Params (см. ниже) для указания стилевых признаков создаваемого окна
procedure FocusControl (Control: TWinControl);	Передает фокус ввода дочернему элементу Control
function GetFormImage: TBitmap;	Содержит текущее изображение окна формы
procedure Next;	Делает активным следующее MDI-окно
procedure Hide;	Скрывает окно
procedure MakeFullyVisible (AMonitor: TMonitor = nil);	Проверяет, полностью ли умещается форма на мониторе, и, при необходимости, изменяет ее положение так, чтобы у нее не было частей на других мониторах
procedure MouseWheelHandler (var Message: TMessage); override;	Вызывается автоматически при получении сообщения от колесика мыши. По умолчанию ничего не делает
procedure Previous;	Делает активным предыдущее MDI-окно
procedure Print;	Печатает окно на принтере
procedure Release;	Ожидает окончания обработки всех событий формы и ее дочерних элементов, после чего удаляет окно и освобождает всю связанную с ним память
procedure SendCancelMode (Sender: TControl);	Восстанавливает начальное состояние окна: освобождает мышь, прекращает прокрутку и закрывает меню
procedure SetFocus;	Передает фокус ввода форме. Форма при этом должна быть активной и видимой
function SetFocusedControl (Control: TWinControl): Boolean; virtual;	Передает фокус ввода указанному элементу
procedure Show;	Показывает форму в немодальном режиме
function ShowModal: Integer;	Показывает форму в модальном режиме и возвращает результат диалога с пользователем
procedure Tile;	Располагает дочерние MDI-окна мозанкой
function WantChildKey (Child: TControl; var Message:	Этот метод вызывается любым размещенным на форме компонентом

продолжение ➤

Таблица 21.3 (продолжение)

Метод	Описание
TMessage): Boolean; virtual;	в момент, когда он получает фокус ввода. Если функция возвращает значение True, весь клавиатурный ввод переназначается формс. По умолчанию возвращает значение False

Структура Params в методе CreateParams описывается следующим типом:

type

```

TCreateParams = record
  Caption: PChar;           // Заголовок окна
  Style: DWORD;             // Стилиевые флаги
  ExStyle: DWORD;           // Дополнительные стилиевые флаги
  X, Y: Integer;            // Координаты левого верхнего угла
  Width, Height: Integer;   // Размеры
  WndParent: HWND;          // Дескриптор владельца
  Param: Pointer;           // Указатель на параметры
                           // для сообщения WM_CREATE
  WindowClass: TWndClass;   // Класс окна
  WinClassName: array[0..63] of Char;
                           // Имя оконного класса
end;
```

Этот метод следует перекрыть, если вы хотите создать окно нестандартного типа. Например, набор возможных значений свойства BorderStyle не предполагает создания окна с «толстой» рамкой, но без заголовка. Однако если в свойство Params.Style поместить значение WS_THICKFRAME **or** WS_POPUP, такое окно будет создано. В проекте Chap_21\MovedWindow\MovWin.dpr создается окно без заголовка, которое, тем не менее, можно перетаскивать мышью:

```
unit Unit1;
```

```
interface
```

```
uses
```

```
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons;
```

```
type
```

```
  TForm1 = class (TForm)
```

```
    BitBtn1: TBitBtn;
```

```
  private
```

```
    { Private declarations }
```

```
  protected
```

```
    // Перехват сообщения WM_NCHITTEST для перемещения
    // окна без заголовка:
```

```

    procedure WMNCHitTest(var Message: TWMNCHitTest);
    message WM_NCHITTEST;
public
    { Public declarations }
    procedure CreateParams(var Params: TCreateParams);
    override;
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.WMNCHitTest(var Message: TWMNCHitTest);
begin
    // Результат HTCAPTION означает, что указатель мыши
    // находится над заголовком. Это заставит Windows
    // перемещать окно:
    Message.Result := HTCAPTION;
end;

procedure TForm1.CreateParams(var Params: TCreateParams);
begin
    inherited CreateParams(Params);
    // Окно с рамкой (WS_THICKFRAME), но без заголовка
    // (WS_POPUP):
    Params.Style := WS_THICKFRAME or WS_POPUP;
end;

end.
```

Сообщение WM_NCHITTEST посылает Windows в момент создания окна при любом изменении его нерабочей области (заголовка, рамки, полос прокрутки и т. п.), а также при перемещении на нем указателя мыши. Обычно программы его не обрабатывают, но в нашем случае обработчик этого события «обманывает» Windows и сообщает операционной системе, что указатель перемещается на области заголовка.

События формы перечислены в табл. 21.4.

Таблица 21.4. События формы

Событие	Описание
property OnActivate: TNotifyEvent;	Возникает в момент активизации окна (при получении им фокуса ввода)

продолжение ➤

Таблица 21.4 (продолжение)

Событие	Описание
type TCanResizeEvent = procedure (Sender: TObject; var NewWidth, NewHeight: Integer; var Resize: Boolean) of object ; property OnCanResize: TCanResizeEvent;	Возникает при изменении размеров окна. Обработчик может указать новые размеры или запретить их изменение, поместив в свойство Resize значение False
TCloseAction = (caNone, caHide, caFree, caMinimize); TCloseEvent = procedure (Sender: TObject; var Action: TCloseAction) of object ; property OnClose: TCloseEvent;	Возникает перед закрытием окна. Параметр Action уточняет необходимые действия: caNone — не закрывать окно; caHide — скрыть окно; caFree — удалить окно; caMinimize — свернуть окно
TCloseQueryEvent = procedure (Sender: TObject; var CanClose: Boolean) of object ; property OnCloseQuery: TCloseQueryEvent;	Возникает перед закрытием окна. В параметре CanClose обработчик сообщает о возможности закрытия окна
property OnDblClick: TNotifyEvent;	Возникает при двойном щелчке на любом расположенном на форме компоненте
property OnDeactivate: TNotifyEvent;	Возникает при передаче активности (фокуса ввода) другому окну
property OnDestroy: TNotifyEvent;	Возникает перед удалением окна
property OnHide: TNotifyEvent;	Возникает перед исчезновением окна
property OnPaint: TNotifyEvent;	Возникает при необходимости прорисовки окна
property OnResize: TNotifyEvent;	Возникает при изменении размеров окна
TShortCutEvent = procedure (var Msg: TWMKey; var Handled: Boolean) of object ; property OnShortCut: TShortCutEvent;	Возникает до события OnKeyDown и предназначено для перескрытия стандартной обработки нажатия клавиш быстрого вызова
property OnShow: TNotifyEvent;	Возникает при появлении окна на экране

Создание и использование форм

Для подключения новой формы к проекту достаточно обратиться к хранилищу объектов и выбрать нужную разновидность формы. Менеджер проекта автоматически подключает новую форму к списку используемых форм и обеспечивает все необходимые действия по ее инициализации. Самая первая подключенная к проекту форма (стандартное имя формы — Form1) становится главным окном программы. Окно этой формы автоматически появляется на экране в момент старта программы. Впрочем, программист может указать любую форму, окно которой

станет главным. Для этого нужно обратиться к команде **Project ► Options**, в открывшемся окне перейти на вкладку **Forms** и в списке **Main form** выбрать нужную форму (рис. 21.2).

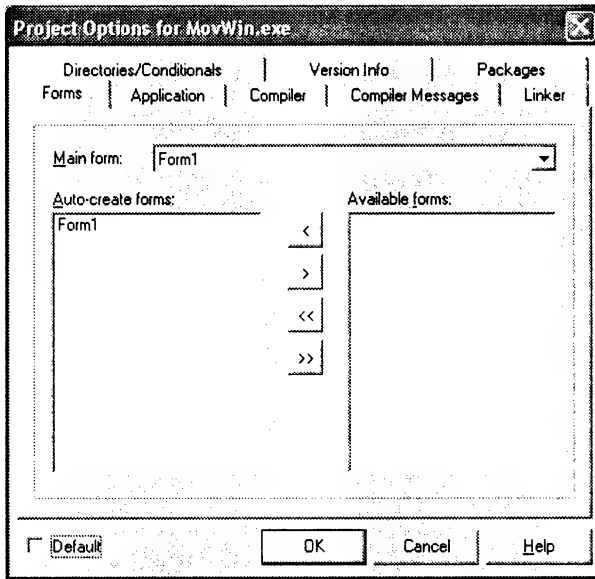


Рис. 21.2. Окно управления проектом

Каждое следующее окно становится видимым только после обращения к его методу **Show** или **ShowModal**. Чтобы обратиться к этим методам, нужно сослаться на объект-окно, который автоматически объявляется в интерфейсном разделе связанного с окном модуля. Для этого, в свою очередь, главное окно должно знать о существовании другого окна, что достигается ссылкой на модуль окна в предложении **Uses**. Если, например, в ходе выполнения одного из методов главного окна программист захочет вызвать окно с именем **fmForm**, связанное с модулем **FormUnit**, он должен сослаться на этот модуль в предложении **Uses** главного окна:

```
implementation
```

```
Uses FormUnit;
```

После этого можно вызвать окно на экран, добавив в программу один из следующих операторов:

```
fmForm.Show;
```

```
fmForm.ShowModal;
```

Delphi автоматизирует вставку ссылки на модуль в предложение **Uses**. Для этого на этапе разработки нужно активизировать главное окно, щелкнув на нем мышью, после чего обратиться к команде **File ► Uses Unit**. В появившемся диалоговом окне (рис. 21.3) нужно выбрать модуль и щелкнуть на кнопке **OK**. Вставляется ссылка в предложение, стоящее за зарезервированным словом **Implementation**, так как обычно главное окно в своей интерфейсной части не ссылается на элементы интерфейсной части второго окна. Точно так же можно при необходимости сослаться

в модуль второго окна на модуль главного окна: активизируйте второе окно и вновь выберите команду **File** ► **Uses Unit**. Замечу, что если программист забудет сослаться на модуль, подключенный к проекту, Delphi при первой же трансляции программы сообщит об этом и предложит вставить недостающую ссылку.

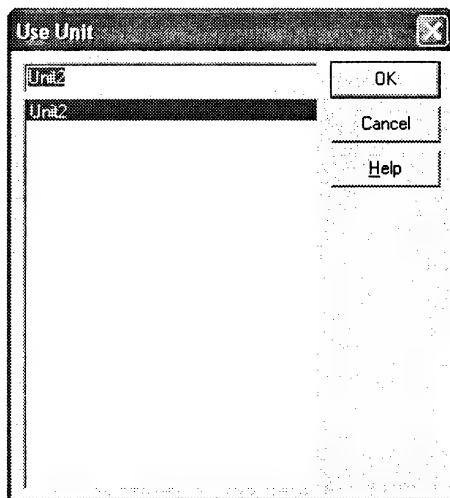


Рис. 21.3. Вставка ссылки на модуль

При вызове метода `Show` второе окно появляется на экране и работает вместе с первым, поэтому управление сразу передается оператору, стоящему за обращением к этому методу. Такие окна называются немодальными, они всегда открываются в одном методе, а закрываются в другом. В отличие от этого обращение к методу `ShowModal` создает модальное окно, которое полностью берет на себя дальнейшее управление программой, поэтому оператор, расположенный следом за обращением к методу `ShowModal` в вызывающей части программы, получит управление только после закрытия модального окна.

Модальные окна часто требуют от пользователя принятия какого-либо решения. С их помощью реализуется диалог с пользователем или создается информационное окно, которое пользователь должен закрыть после ознакомления с содержащейся в нем информацией. Если от пользователя требуется принятие решения, в модальное окно вставляются флажки, переключатели, кнопки и другие интерфейсные элементы, с помощью которых пользователь может сообщить программе о принятом решении. В момент завершения диалога модальное окно должно поместить число, соответствующее решению пользователя, в свое свойство `ModalResult`. Некоторые стандартные кнопки (`OK`, `Yes`, `No`, `Cancel` и т. п.) автоматически выполняют эти действия: помещают нужное число в свойство `ModalResult` и закрывают окно. В других случаях об этом должен позаботиться программист. Вызывающая программа получает значение свойства `ModalResult` как возвращаемое значение функции `ShowModal` и может тут же его проанализировать:

```
if Form2.ShowModal = mrXXX then ...
```

Возможен и такой вариант:

```
Form2.ShowModal;
if Form2.ModalResult = mrXXX then ...
```

Для закрытия окна (модального или немодального) используется метод Hide или Close. Следует учесть, что метод Close всегда помещает в свойство ModalResult значение 2 (mrCancel), в то время как метод Hide не меняет значения этого свойства, поэтому, если программист хочет передать в вызывающую программу нестандартный модальный результат, следует ввести в программу следующие операторы:

```
ModalResult := MyResult;
Hide; // Но ни в коем случае Close!
```

По умолчанию среда Delphi настроена так, что при подключении нового окна к проекту менеджер проекта размещает его имя в списке автоматически создаваемых окон (на рис. 21.2 это список Auto-Create forms). В этом случае программист может не заботиться об инициализации соответствующего объекта окна. Однако если в программе используется множество окон, их автоматическое создание в момент старта программы может существенно затянуть процесс ее загрузки. В то же время немедленное создание всех окон вовсе не обязательно, так как вряд ли вам понадобится одновременно показывать их на экране в немодальном режиме. Профессиональные программисты никогда не создают все оконные объекты в момент старта программы, но используют их по мере надобности. Для этого снимается флажок Auto create forms & data modules на вкладке Designer окна Environment Options (открывается командой Tools ► Environment Options) или в окне менеджера проекта (см. рис. 21.2) ссылки на эти формы переносятся в список Available forms, а обращение к окну реализуется так:

```
if not Assigned(Form2) then                                // Проверяем, создан ли
                                                                // оконный объект
    Form2 := TForm2.Create(Self);                             // Нет — создаем его
if Form2.ShowModal = mrXXX then ... // и используем
```

Глава 22

Программа

Конечной целью программиста, работающего в Delphi, является создание *программы* — исполняемого файла, который может загружаться и выполняться под управлением 32-разрядной версии Windows. В этой главе рассматриваются вопросы, относящиеся к программе в целом. В ней, в частности, описываются многочисленные вспомогательные файлы, связанные с программой, доступные программе глобальные объекты, ее динамическая настройка с помощью файлов инициализации и/или системного реестра. В конце главы приводятся некоторые сведения о 32-разрядной версии Windows и механизмах взаимодействия программы с этой операционной системой.

Программные файлы

Delphi создает множество файлов, связанных с одной программой¹. Перед тем как начать их рассмотрение, хочу порекомендовать следовать золотому правилу: для каждого проекта выделять отдельную папку и хранить в ней все относящиеся к проекту файлы; это значительно упростит перенос проекта в другую папку, например, на гибкий диск и затем с него на другую машину.

Файл проекта

Файл проекта имеет расширение DPR. Обычно он не виден в окне кода — чтобы его увидеть, следует воспользоваться командой **Project ▸ View source** среды Delphi. Если закрыть файл проекта с помощью команды **File ▸ Close** или клавиш **Alt+F4**, вместе с ним закроются и все остальные формы проекта. Чтобы этого не произошло, используйте клавиши **Ctrl+F4** или команду **Close Page** контекстного меню файла проекта.

Файл проекта содержит код главной программы, с помощью которого создается объект главной формы программы и, возможно, объекты некоторых других форм, а также обеспечивается связь программы с ядром Windows. Ниже показан типичный пример файла проекта, содержащего единственное окно формы.

¹ Речь идет о полноценной Windows-программе; при разработке консольных приложений, библиотек DLL, собственных модулей и в некоторых других специальных случаях может быть создан единственный файл.

```
program Project1;  
  
uses  
    Forms,  
    Unit1 in 'Unit1.pas' {Form1};  
  
{$R *.RES}  
  
begin  
    Application.Initialize;  
    Application.CreateForm(TForm1, Form1);  
    Application.Run;  
end.
```

Зарезервированное слово **program** открывает файл проекта; оно может встретиться в программе лишь один раз.

В предложении **uses** перечисляются все (или лишь некоторые) связанные с проектом формы. Помимо стандартного модуля *Forms*, необходимого для создания главного окна программы, в нем указывается также модуль *Unit1*, связанный собственно с главным окном. Обратите внимание на использование зарезервированного слова **in** для указания файла с текстом модуля ('Unit1.pas') и комментария ({*Form1*}), который именуется объект-окно, создаваемый модулем *Unit1*. Такого рода объявления Delphi автоматически создает для каждого включенного в проект модуля. Только перечисленные в этом предложении модули Delphi считает входящими в проект, и их алфавитный список появляется при выборе команды View ► Units; а при выборе команды View ► Forms отображается список всех перечисленных в комментариях объектов.

Следует учесть, что если вы хотите перенести проект в другую папку и при этом сохранить его работоспособность, нужно сначала с помощью команды File ► Save Project As скопировать в эту папку файл проекта, а затем с помощью команды File ► Save As перенести туда все связанные с проектом модули: только в этом случае Delphi сумеет внести необходимые корректировки в файл проекта. Но если все файлы хранятся в единственной папке, в предложении **uses** не указываются пути доступа, и вы можете безболезненно разом скопировать все файлы в другую папку.

ПРИМЕЧАНИЕ

Настроечный DSK-файл, в котором среда сохраняет информацию о состоянии экрана в момент выхода из Delphi, также содержит полные пути доступа к открытым файлам. При переносе проекта этот файл не следует копировать.

В разделе исполняемых операторов формируются, как минимум, три оператора: вызовы методов *Initialize*, *CreateForm* и *Run* глобального объекта *Application*. Метод *Initialize* предусмотрен «на всякий случай» и по умолчанию ничего не делает. Чтобы заставить его работать, следует поместить указатель на соответствующую процедуру без параметров в глобальную переменную *InitProc*. Это можно сделать в секции инициализации любого модуля (напомню, что код

этих секций активизируется до начала работы основной программы) или непосредственно в теле главного проекта:

```
procedure Init;
begin
    // Выполняет какую-то работу;
end;

begin
    InitProc := @Init;
    Application.Initialize; // Обращение к Init
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.
```

Разумеется, в этом последнем случае проще в первом операторе основной программы явно обратиться к процедуре Init. Вообще говоря, в подавляющем большинстве случаев оператор Application.Initialize можно удалить без каких-либо последствий для программы.

Следует учесть, что до оператора Application.Run можно вставить любые другие операторы Delphi. В приводимом ниже примере (он заимствован из справочной службы Delphi) в программе создается пять окон. В форме Form5 имеется компонент ProgressBar1, с помощью которого визуализируется процесс загрузки программы, точнее — процесс создания остальных окон.

```
begin
    with TForm5.Create(nil) do
        try
            ProgressBar1.Max := 100;
            Show; // Показываем форму Form5 с индикатором
                // ProgressBar
            Update; // Прорисовываем форму Form5
            Application.CreateForm(TForm1, Form1);
            ProgressBar1.StepBy(25);
            Application.CreateForm(TForm2, Form2);
            ProgressBar1.StepBy(25);
            Application.CreateForm(TForm3, Form3);
            ProgressBar1.StepBy(25);
            Application.CreateForm(TForm4, Form4);
            ProgressBar1.StepBy(25);
        finally
            Free; // Удаляем ненужную форму Form5
        end;
    Application.Run;
end.
```

С помощью метода Application.Run программа подключается к циклу обработки сообщений Windows. Выход из метода возможен лишь после закрытия главного окна программы.

Программист может показать диалоговое окно с запросом пароля и блокировать вызов метода `Application.Run`, если введенный пользователем пароль неверен. В следующем примере (листинг 22.1) в проекте используются две формы: стандартная форма `InputQuery` и обычная главная форма `MainForm`. Форма `InputQuery` создается при обращении к одноименной функции, определенной в модуле `Dialogs`. Она представляет собой небольшое диалоговое окно с однострочным текстовым полем `TEdit` и двумя кнопками — `OK` и `Cancel`. В окне пользователь должен ввести пароль (Delphi) и нажать клавишу `Enter` (проект `Chap_22\Password\Password.dpr`).

Листинг 22.1. Защита программы паролем¹

```

program Password;

uses
  Forms,
  Dialogs, // В этом модуле определена функция InputQuery
  Unit1 in 'Unit1.pas' {MainForm};

{$R *.RES}
var
  Passwrd: String;
begin
  // Запрашиваем пароль:
  if InputQuery('Окно ввода пароля',
    'Введите пароль:',Passwrd) then
    // Проверяем его:
    if Passwrd='Delphi' then
      begin // Все в порядке, пароль верен
        Application.CreateForm(TMainForm, MainForm);
        Application.Run;
      end else
        ShowMessage('Пароль не верен!');
  end.

```

Таким же способом можно создавать пробные версии программ, которые будут функционировать только до определенной даты или до исчерпания заданного количества запусков (см. проект `Chap_22\Registry\Trial.dpr`).

С проектом может быть не связано ни одного видимого окна. Такие программы называются *консольными*. Они используются для проведения работ, не требующих диалога с пользователем и соответствующих средств ввода-вывода информации. Например, они могут пассивно ждать наступления определенного часа (обычно ночью), чтобы провести резервное копирование базы данных.

Консольные программы не вызывают методы `CreateForm` и `Run` объекта `Application`. Вместо этого в раздел исполняемых операторов файла проекта помещается код, который реализует необходимые действия.

¹ Все представленные в книге тексты программ можно загрузить с сайта издательства по адресу <http://www.piter.com/download> — *Примеч. ред.*

Для создания консольного приложения нужно выбрать значок **Console Application** на вкладке **New** окна хранилища объектов. В ответ Delphi создаст файл проекта следующего вида:

```
program Project1;
{$APPTYPE CONSOLE}
uses SysUtils;

begin
    // Insert user code here
end.
```

Директива `{ $APPTYPE CONSOLE }` дает в распоряжение программы «консольные» средства ввода-вывода. В этом случае программа автоматически выполняется в окне MS-DOS, и с помощью операторов `ReadLn`, `WriteLn` программист может ввести что-то с клавиатуры и вывести на экран так, как если бы программа работала под управлением MS-DOS. Если программе не нужны даже такие средства общения с внешним миром, эту директиву следует удалить.

Несмотря на полное отсутствие видимых окон или их замену окном MS-DOS, консольная программа остается полноценным 32-разрядным приложением и после ссылки на нужные модули ей доступны глобальные объекты, средства типа `InputQuery`, `ShowMessage` и т. п.

Файлы модулей

Файлы с текстами модулей имеют расширения **PAS**. После компиляции для каждого такого файла создается одноименный файл машинного кода с расширением **DCU**. Если модуль содержит объект-окно, на этапе конструирования создается и автоматически изменяется файл описания формы с расширением **DFM** (файл будет реально создан после сохранения модуля на диске). Файлы **PAS** и **DFM** жизненно важны для проекта: удаление любого из них приведет к невозможности повторной компиляции проекта.

В ранних версиях Delphi файл **DFM** был двоичным, то есть он описывал форму на языке, понятном только компилятору. В версиях 5...7 он может быть текстовым. Например:

```
object Form1: TForm1
    Left = 272
    Top = 107
    Width = 544
    Height = 375
    Caption = 'Form1'
    Color = clBtnFace
    Font.Charset = DEFAULT_CHARSET
    Font.Color = clWindowText
    Font.Height = -11
    Font.Name = 'MS Sans Serif'
    Font.Style = []
    OldCreateOrder = False
```

```

PixelsPerInch = 96
TextHeight = 13
object Button1: TButton
    Left = 32
    Top = 24
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 0
end
end

```

В таком формате программист может вручную отредактировать файл описания формы. Если, например, в предыдущем примере вместо типа `TButton` написать `TLabel`, после загрузки проекта в среду Delphi на месте кнопки окажется метка. Выбор формата файла описания формы осуществляется с помощью флажка **New forms as text** на вкладке **Designer** диалогового окна настройки параметров среды (открывается командой **Tools** ► **Environment Options**). Поскольку, как уже говорилось, файл описания формы жизненно важен для функционирования проекта, ручное его изменение может привести к ошибкам. В этом случае восстановить его можно, удалив из него все внесенные вами изменения, либо воспользовавшись его резервной копией, имеющей расширение `~DF`.

В версии 7 помимо файлов с расширениями `PAS` и `DFM` создается также `DDP`-файл, в котором сохраняются сведения обо всех созданных для окна диаграммах.

Файлы ресурсов

Файл проекта содержит директиву компилятора `{ $R *.RES }`, с помощью которой на этапе компоновки к исполняемому файлу присоединяется файл ресурсов. Последний имеет имя, совпадающее с именем проекта, и расширение `RES`. Он создается автоматически для каждого проекта. В него Delphi помещает значок программы, номер ее версии и тому подобные данные. В этом файле не следует размещать никаких дополнительных ресурсов (указателей, строк, значков и т. п.), так как при каждой компиляции файл ресурсов создается заново. Если в программу необходимо ввести дополнительные ресурсы, их нужно разместить в отдельном файле и вставить в файл проекта соответствующую директиву (пример создания дополнительного ресурса описан в разделе «Указатели мыши» главы 14).

Файлы настройки

Для проекта создаются также три вспомогательных текстовых файла, в которых сохраняются параметры настройки компилятора (`CFG`), проекта (`D0F`) и среды (`DSK`). Хотя все они текстовые, их ручное редактирование нежелательно, так как при внесении в них неправильных изменений компиляция проекта может сопровождаться сообщением «Программа выполнила недопустимую операцию и будет закрыта», после чего Delphi выгружается из памяти. В такого рода случаях следует просто удалить все три настроечных файла и повторить компиляцию — Delphi создаст новые настроечные файлы с параметрами настройки по умолчанию.

Файлы резервных копий

При внесении в проект изменений Delphi обычно создает резервные копии измененных файлов. Эти копии имеют расширения, начинающиеся символом тильда (~). Они создаются «на всякий случай» и содержат состояние проекта и/или модулей до внесения изменений. От создания резервных копий можно отказаться, если снять флажок **Create backup file** на вкладке **Display** диалогового окна **Editor Options** (открывается командой **Tools ▶ Editor Options**).

Глобальные объекты

С любой запущенной программой автоматически связываются от двух до пяти глобальных объектов: **Application** (Программа), **Screen** (Экран), **Printer** (Принтер), **Session** (Сеанс) и **Clipboard** (Буфер обмена). Все они объявлены как глобальные переменные в наиболее часто используемых модулях: **Application** и **Screen** — в модуле **Forms**, **Printer** — в модуле **Printers**, **Session** — в модуле **DBTables** и, наконец, **Clipboard** — в модуле **Clipbrd**. Создание этих объектов реализуется в секциях инициализации соответствующих модулей, так что к моменту начала исполнения первого оператора файла проекта они уже готовы к работе.

Application

Объект **Application** относится к классу **TApplication** и инкапсулирует важнейшие свойства и методы программы как таковой. Он создается автоматически и доступен любой программе¹. Объекта **Application** нет в палитре компонентов, поэтому его свойства недоступны на этапе конструирования, но становятся доступными при прогоне программы².

Объект **Application** является посредником между программой и операционной системой **Windows**. В частности, с его помощью осуществляется диспетчеризация сообщений **Windows**, реализуется контекстная справочная служба, он обрабатывает клавиши быстрого доступа, исключительные ситуации и т. д. Свойства класса **TApplication** перечислены в табл. 22.1.

Таблица 22.1. Свойства класса **TApplication**

Свойство	Описание
property Active : Boolean;	Содержит значение True , если любое окно программы имеет фокус ввода
property AutoDragDocking : Boolean;	Если содержит значение True , процедура «причаливания» (Drag&Dock) реализуется автоматически, в противном случае — только при перемещении указателя мыши с нажатой клавишей Ctrl

¹ Для обычных **Windows**-приложений файл проекта ссылается на модуль **Forms**, поэтому объекты **Application** и **Screen** действительно всегда доступны.

² Некоторые атрибуты программы (в том числе ее значок, сведения о версии и используемом языке) могут устанавливаться с помощью менеджера проекта (команда **Project ▶ Options**, вкладка **Application**).

Свойство	Описание
property CurrentHelpFile: String;	Определяет имя текущего справочного файла
property ExeName: String;	Содержит полное имя (с маршрутом доступа) исполняемого файла программы
property HelpFile: String;	Определяет имя файла справочной службы
property HelpSystem: IHelpSystem;	Открывает доступ к интерфейсу справочной системы для непосредственного взаимодействия с менеджером справочной службы
property Hint: String;	Содержит длинную часть оперативной подсказки
property HintColor: TColor;	Определяет цвет фона окна оперативной подсказки
property HintHidePause: Integer;	Определяет паузу (в миллисекундах) от момента появления оперативной подсказки до ее исчезновения
property HintPause: Integer;	Определяет паузу (в миллисекундах) от момента остановки указателя мыши до появления оперативной подсказки
property HintShortCuts: Boolean;	Если содержит значение True, текст оперативной подсказки автоматически дополняется информацией о клавишах быстрого доступа
property HintShortPause: Integer;	Определяет паузу (в миллисекундах) появления оперативной подсказки при переходе с одного компонента на другой
property Icon: TIcon;	Определяет значок программы
property MainForm: TForm;	Определяет главное окно программы
property ShowHint: Boolean;	Разрешает/запрещает показ оперативной подсказки для всех окон программы
property ShowMainForm: Boolean;	Разрешает/запрещает показ главного окна в момент старта программы
property Terminated: Boolean;	Содержит значение True, если программа должна прекратить работу. Используется в приложениях с интенсивным счетом для контроля необходимости прекращения дальнейшей работы
property Title: String;	Определяет текст на кнопке свернутой программы
property UpdateFormatSettings: Boolean;	Если содержит значение True, программа автоматически будет учитывать новые параметры формата, установленные в Windows (например, новый разделитель целой и дробной частей числа) при их изменении в момент работы программы

продолжение ➤

Таблица 22.1 (продолжение)

Свойство	Описание
property UpdateMetricSettings: Boolean;	Если содержит значение True, программа автоматически будет учитывать новые метрические установки Windows (например, новый системный шрифт) при их изменении в момент работы программы

Методы класса TApplication перечислены в табл. 22.2.

Таблица 22.2. Методы класса TApplication

Метод	Описание
procedure ActivateHint (CursorPos: TPoint);	Показывает оперативную подсказку в заданной координатами CursorPos позиции экрана
procedure BringToFront;	Помещает окно, которое было активным в последний раз, поверх остальных окон
procedure CancelHint;	Запрещает показ оперативной подсказки
procedure CreateForm (FormClass: TFormClass; var Reference);	Создает объект-окно: вызывает конструктор класса TFormClass и связывает созданный с его помощью объект с переменной Reference
procedure HandleException (Sender: TObject);	Осуществляет обработку исключительных ситуаций, заданную по умолчанию. Используется при разработке новых компонентов
procedure HandleMessage;	Приостанавливает работу программы до тех пор, пока не будут обработаны все сообщения, после чего генерирует событие OnIdle
function HelpCommand(Command: Word; Data: LongInt): Boolean;	Посылает справочной службе команду Command и параметр Data
function HelpContext(Context: THelpContext): Boolean;	Вызывает раздел Context справочной службы и возвращает значение True, если такой раздел существует
function HelpJump(const JumpID: String): Boolean;	Вызывает раздел справочной службы по его идентификатору JumpID и возвращает значение True, если такой раздел существует
procedure HideHint;	Убирает текущую оперативную подсказку
function MessageBox(Text, Caption: PChar; Flags: LongInt): Integer;	Создает и показывает диалоговое окно с текстом и кнопками: Text — сообщение в окне; Caption — заголовок окна. Назначение параметра Flags и возвращаемое значение описаны ниже
procedure Minimize;	Свертывает программу до кнопки на панели задач

Метод	Описание
procedure NormalizeTopMosts;	Приостанавливает действие стиля fsTopMost (поместить окно поверх остальных) для всех окон программы, кроме главного
procedure NormalizeAllTopMosts;	Распространяет действие метода NormalizeTopMosts на главное окно программы
procedure ProcessMessages;	То же, что метод HandleMessage, но событие OnIdle не генерируется
procedure Restore;	Восстанавливает первоначальные размеры всех окон программы, имевшие место до ее свертывания
procedure RestoreTopMosts;	Восстанавливает действие стиля fsTopMost для всех окон программы
procedure Run;	Запускает цикл получения и обработки Windows-сообщений
procedure ShowException (E: Exception);	Реализует обработку исключительных ситуаций, заданную по умолчанию
procedure Terminate;	Завершает работу программы

С помощью методов HandleException и ProcessMessages программа может временно приостановить свою работу до тех пор, пока не будут обработаны все предназначенные ей сообщения Windows. Это бывает необходимо для обновления видимых компонентов в процессе отображения длительного по времени цикла работы. Если, например, программа изменит текст метки (компонента TLabel) или состояние индикатора процесса (компонента TProgressBar), сообщение об этом поступит в очередь сообщений и останется в ней, пока не закончится текущая работа. Вызов метода HandleException или ProcessMessages в этом случае позволит метке или индикатору немедленно обновить свой вид и отобразить новую фазу процесса. Разница между методами HandleException и ProcessMessages состоит в том, что первый после очистки очереди сообщений вызывает обработчик фонового события OnIdle, а второй не делает этого (о назначении события OnIdle см. ниже).

Функция MessageBox дает удобное средство вывода на экран небольшого сообщения. Параметр Flags может быть произвольной комбинацией следующих значений:

- mb_AbortRetryIgnore — в окно вставляются кнопки Abort, Retry, Ignore;
- mb_ApplModal — пользователь должен закрыть это окно перед тем, как продолжить работу с окном, его породившим; однако он может переходить в другие окна и работать с ними (этот признак устанавливается по умолчанию, если не определены ни mb_SystemModal, ни mb_TaskModal);
- mb_DefButton1 — кнопкой по умолчанию является кнопка 1;
- mb_DefButton2 — кнопкой по умолчанию является кнопка 2;
- mb_DefButton3 — кнопкой по умолчанию является кнопка 3;

- `mb_DefButton4` — кнопкой по умолчанию является кнопка 4;
- `mb_Help` — в окно вставляется кнопка Help;
- `mb_IconAsterisk` — в окно вставляется значок в виде строчной буквы *i* в кружке;
- `mb_IconExclamation` — в окно вставляется значок в виде восклицательного знака;
- `mb_IconHand` — в окно вставляется значок в виде знака *Смон*;
- `mb_IconInformation` — то же, что и `mb_IconAsterisk`;
- `mb_IconQuestion` — в окно вставляется значок в виде знака вопроса;
- `mb_IconStop` — то же, что и `mb_IconHand`;
- `mb_Ok` — в окно вставляется кнопка OK;
- `mb_OkCancel` — в окно вставляются кнопки OK и Cancel;
- `mb_RetryCancel` — в окно вставляются кнопки Retry и Cancel;
- `mb_SystemModal` — все программы приостанавливают свою работу, пока пользователь не закроет окно;
- `mb_TaskModal` — приостанавливается работа во всех окнах данной программы до тех пор, пока пользователь не закроет окно;
- `mb_YesNo` — в окно вставляются кнопки Yes, No;
- `mb_YesNoCancel` — в окно вставляются кнопки Yes, No, Cancel.

Пример:

```
Application.MessageBox('Текст в окне', 'Заголовок окна',
                        mb_IconQuestion + mb_YesNo);
```

Функция `MessageBox` возвращает следующие значения:

- `idAbort` — была нажата кнопка Abort;
- `idCancel` — была нажата кнопка Cancel;
- `idIgnore` — была нажата кнопка Ignore;
- `idOk` — была нажата кнопка OK;
- `idNo` — была нажата кнопка No;
- `idRetry` — была нажата кнопка Retry;
- `idYes` — была нажата кнопка Yes.

Альтернативой функции `MessageBox` является глобальная функция `MessageDlg`, объявленная в модуле `Dialogs`:

```
function MessageDlg(const Msg: String; AType: TMsgDlgType;
  AButtons: TMsgDlgButtons; HelpCtx: Longint): Word;
```

Эта функция имеет параметр `HelpCtx`, с помощью которого создаваемое ею окно можно легко связать со справочной службой. В отличие от окна `MessageBox`, заголовок окна `MessageDlg` не может изменяться программистом — он определяется типом окна, задаваемым параметром `AType` со следующими возможными значениями:

- `mtWarning` — значок в виде восклицательного знака, заголовок `Warning`;
- `mtError` — значок в виде знака *Смон*, заголовок `Error`;
- `mtInformation` — значок в виде буквы *i*, заголовок `Information`;
- `mtConfirmation` — значок в виде вопросительного знака, заголовок `Confirm`;
- `mtCustom` — нет значка, заголовок содержит имя исполняемого файла программы.

Набор кнопок задается в виде конструктора множества и может включать:

- `mbYes` — кнопка `Yes` (результат, возвращаемый функцией при щелчке на этой кнопке — `mrYes`);
- `mbNo` — кнопка `No` (`mrNo`);
- `mbOk` — кнопка `OK` (`mrOk`);
- `mbCancel` — кнопка `Cancel` (`mrCancel`);
- `mbHelp` — кнопка `Help` (после щелчка на этой кнопке окно не закрывается);
- `mbAbort` — кнопка `Abort` (`mrAbort`);
- `mbIgnore` — кнопка `Ignore` (`mrIgnore`);
- `mbAll` — кнопка `All` (`mrAll`).

В отличие от окна `MessageBox`, раскрытие окна `MessageDlg` не сопровождается системным звуком.

События класса `TApplication` представлены в табл. 22.3.

Таблица 22.3. События класса `TApplication`

Событие	Описание
property <code>OnActivate: TNotifyEvent;</code>	Возникает, когда программа получает фокус ввода
property <code>OnDeactivate: TNotifyEvent;</code>	Возникает, когда программа теряет фокус ввода
<code>TExceptionEvent = procedure (Sender: TObject; E: Exception) of object;</code> property <code>OnException: TExceptionEvent;</code>	Обработчик этого события реализует обработку исключительных ситуаций, заданную по умолчанию
<code>THelpEvent = function (Command: Word; Data: LongInt; var CallHelp: Boolean): Boolean of object;</code> property <code>OnHelp: THelpEvent;</code>	Возникает при каждом обращении к справочной службе
property <code>OnHint: TNotifyEvent;</code>	Возникает при необходимости отобразить длинную часть оперативной подсказки
<code>TIdleEvent = procedure (Sender: TObject; var Done: Boolean) of object;</code> property <code>OnIdle: TIdleEvent;</code>	Обработчик этого события реализует фоновую работу (см. пояснения ниже)

— продолжение ➤

Таблица 22.3 (продолжение)

Событие	Описание
<code>TMessageEvent = procedure (var Msg: TMsg; var Handled: Boolean) of object; property OnMessage: TMessageEvent;</code>	Возникает при получении программой сообщения Windows. С помощью обработчика этого события можно обработать несколько или все сообщения до того, как их получит активная форма
<code>property OnMinimize: TNotifyEvent;</code>	Возникает при свертывании программы
<code>property OnRestore: TNotifyEvent;</code>	Возникает при восстановлении программы после ее свертывания
<code>property OnShowHint: TShowHintEvent;</code>	Возникает при необходимости отобразить окно с оперативной подсказкой

Событие `OnIdle` возникает всякий раз, когда программа обнаруживает, что ее очередь сообщений пуста. Обработчик этого события может незаметно для пользователя выполнить вспомогательную работу, например, проверить орфографическую правильность только что введенного слова или собрать и обработать статистические данные. Важно помнить, что выполняемая в обработчике работа не может занимать много времени, иначе программа перестанет реагировать на запросы пользователя. Если фоновая работа отнимает много времени, ее следует разбить на сравнительно небольшие фрагменты или периодически вызывать в ней метод `HandleMessage`. Обработчик в параметре `Done` сообщает о необходимости продолжить фоновую работу (`Done = False`) или о ее завершении (`Done = True`). В первом случае программа просмотрит очередь сообщений и после ее очистки вернет управление обработчику `OnIdle`. Во втором случае программа будет ожидать поступления новых сообщений.

СОВЕТ Событие `OnIdle` введено в Delphi для совместимости с платформой Windows 3.x, в которой не поддерживается многопоточность. Если вы работаете с 32-разрядной версией Windows и вам действительно нужно проделать некоторую фоновую работу, имеет смысл поручить ее выполнение отдельному потоку команд (см. ниже подраздел «Процессы и потоки» в разделе «Программа и Windows»).

Screen

Объект `Screen` класса `TScreen` инкапсулирует свойства и методы, упрощающие работу с дисплеем компьютера, в том числе с его помощью оперативно изменяется вид используемого программой указателя мыши. Как правило, программа устанавливает указатель в виде песочных часов перед началом длинной по времени работы и восстанавливает заданную по умолчанию форму указателя после ее завершения. Для этого используется свойство `Cursor` объекта `Screen`:

```
Screen.Cursor := crHourGlass;
// Выполняем длительную работу:
...
// Восстанавливаем форму, заданную по умолчанию:
```

```
Screen.Cursor := crDefault;
```

Наиболее важные свойства объекта Screen перечислены в табл. 22.4.

Таблица 22.4. Наиболее важные свойства объекта Screen

Свойство	Описание
property ActiveControl: TWinControl;	Содержит ссылку на элемент с фокусом ввода
property ActiveForm: TForm;	Содержит ссылку на окно с фокусом ввода
property Cursor: TCursor;	Определяет текущий указатель мыши для окон программы
property Cursors[Index: Integer]: HCursor;	Открывает доступ к одному из зарегистрированных указателей мыши по его индексу
property DefaultKbLayout: HKL;	Содержит исходную раскладку клавиатуры (см. пояснения ниже)
property DesktopHeight: Integer;	Содержит высоту экрана относительно высоты экрана первичного монитора
property DesktopLeft: Integer;	Содержит горизонтальное положение левого верхнего угла экрана относительно такого же параметра экрана первичного монитора
property DesktopTop: Integer;	Содержит вертикальное положение левого верхнего угла экрана относительно такого же параметра экрана первичного монитора
property DesktopWidth: Integer;	Содержит ширину экрана относительно ширины экрана первичного монитора
property Fonts: TStrings;	Содержит список имен всех экранных шрифтов
property FormCount: Integer;	Содержит общее количество показываемых на экране окон программы
property Forms[Index: Integer]: TForm;	Открывает доступ к окну программы по его индексу
property Height: Integer;	Содержит высоту экрана в пикселах
property HintFont: TFont;	Содержит шрифт для оперативной подсказки
property IconFont: TFont;	Определяет шрифт для надписи на кнопке свернутой программы
property MenuFont: TFont;	Содержит шрифт для команд меню
property MonitorCount: Integer;	Содержит количество мониторов
property Monitors[Index: Integer]: TMonitor;	Открывает индексированный доступ к монитору
property PixelsPerInch: Integer;	Содержит разрешающую способность экрана в пикселах на линейный дюйм
property Width: Integer;	Содержит ширину экрана в пикселах

Программа с помощью API-функции `ActivateKeyboardLayout` может изменить раскладку клавиатуры. Свойство `DefaultKbLayout` позволяет восстановить прежнюю раскладку. Следующая программа всегда запускается с русской язычной раскладкой клавиатуры:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1}, Windows;

{$R *.res}

begin
  if Screen.DefaultKbLayout = 67699721 then
    ActivateKeyboardLayout (HKL_NEXT, KLF_REORDER);
  Application.CreateForm(TForm1, Form1);
  Application.Run;
  ActivateKeyboardLayout (Screen.DefaultKbLayout, KLF_REORDER);
end.
```

При работе под управлением Windows 2000/XP программа может работать одновременно с множеством мониторов, которые в совокупности образуют первичный монитор, составленный из остальных слева направо и сверху вниз. Ряд свойств `DesktopXXX` определяют положение и размеры текущего монитора в координатах первичного. Если программа работает с единственным монитором, значения этих свойств и свойств `XXX` (`Height`, `Left`, `Top`, `Width`) совпадают. Свойство `MonitorCount` содержит общее количество обслуживаемых мониторов; с помощью свойства `Monitors` можно указать конкретный монитор в рамках первичного.

Свойство `Fonts` содержит список всех зарегистрированных в системе экраных шрифтов. Этот список можно связать с раскрывающимся списком, чтобы пользователь мог выбрать нужный шрифт:

```
ComboBox1.Items.Assign(Screen.Fonts);
```

С объектом `Screen` связаны два события:

- **property** `OnActiveControlChange`: `TNotifyEvent`; — возникает при переходе фокуса ввода от одного элемента к другому;
- **property** `OnActiveFormChange`: `TNotifyEvent`; — возникает при переходе фокуса ввода от одной формы к другой.

Printer

Объект `Printer` автоматически создается, если в программе указана ссылка на модуль `Printers`. Этот объект предоставляет программисту все необходимое для того, чтобы заставить программу выводить данные на один из подключенных к компьютеру принтеров.

Вывод на принтер в Windows ничем не отличается от вывода на экран: в распоряжение программиста предоставляется свойство `Canvas` объекта `Printer`, содержащее набор графических инструментов и методы, свойственные классу `TCanvas` (подробнее см. раздел «Графический инструментарий» в главе 13). Размер листа бумаги в пикселах принтера определяют свойства `Height` и `Width`, а набор принтерных шрифтов — свойство `Fonts`.

Свойства объекта `Printer` перечислены в табл. 22.5.

Таблица 22.5. Свойства объекта `Printer`

Свойство	Описание
property <code>Aborted: Boolean;</code>	Содержит значение <code>True</code> , если выполнение предыдущего задания на печать было досрочно прервано
property <code>Canvas: TCanvas;</code>	Канва принтера — основной инструмент создания изображения на листе бумаги
type <code>TPrinterCapability = (pcCopies, pcOrientation, pcCollation);</code> property <code>Capabilities: TPrinterCapabilities;</code>	Содержит текущие параметры драйвера принтера: <code>pcCopies</code> — может печататься несколько копий документа; <code>pcOrientation</code> — драйвер поддерживает разную ориентацию листа бумаги; <code>pcCollation</code> — документ печатается в подбор по экземплярам
property <code>Copies: Integer;</code>	Указывает требуемое количество копий документа
property <code>Fonts: TStrings;</code>	Содержит список всех доступных принтеру шрифтов
<code>TPrinterOrientation = (poPortrait, poLandscape);</code> property <code>Orientation: TPrinterOrientation;</code>	Определяет ориентацию листа бумаги: <code>poPortrait</code> — вертикальная; <code>poLandscape</code> — горизонтальная
property <code>PageHeight: Integer;</code>	Содержит высоту листа бумаги в пикселах принтера
property <code>PageNumber: Integer;</code>	Содержит номер печатаемой страницы документа (начинается с 1)
property <code>PageWidth: Integer;</code>	Содержит ширину листа бумаги в пикселах принтера
property <code>PrinterIndex: Integer;</code>	Содержит индекс принтера, назначенного принтером по умолчанию в списке <code>Printers</code> окна параметров печати
property <code>Printers: TStrings;</code>	Содержит список всех доступных принтеров
property <code>Printing: Boolean;</code>	Содержит значение <code>True</code> , если принтер занят печатью документа
property <code>Title: String;</code>	Содержит имя задания на печать

Методы объекта `Printer` перечислены в табл. 22.6.

Таблица 22.6. Методы объекта `Printer`

Метод	Описание
procedure <code>Abort;</code>	Досрочно прекращает печать документа
procedure <code>BeginDoc;</code>	Начинает печать документа
procedure <code>EndDoc;</code>	Завершает печать документа
procedure <code>NewPage;</code>	Начинает печать новой страницы документа

Существует множество способов печати текста на принтере. Прежде всего следует упомянуть о глобальной процедуре `AssignPrn` (она определена в модуле `Printers`), позволяющей использовать принтер как текстовый файл и печатать текстовые строки с помощью процедуры `WriteLn`. В листинге 22.2 приведен полный текст модуля, на форме которого расположено многострочное текстовое поле `Memol` и 4 кнопки: для выбора текстового файла и ввода его содержимого в поле, для выбора нужного шрифта отображения/печати документа, для инициирования процесса печати и для завершения работы программы (проект `Chap_22\Print of Text (AssignPrn)\PrintText.dpr`).

Листинг 22.2. Печать текста с помощью процедуры `AssignPrn`

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons;

type
  TForm1 = class(TForm)
    Memol: TMemo;
    Button1: TButton;
    Button2: TButton;
    OpenFileDialog1: TOpenDialog;
    BitBtn1: TBitBtn;
    Button3: TButton;
    FontDialog1: TFontDialog;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

```

```

var
  Form1: TForm1;

implementation

uses Printers;  // Эта ссылка обязательна!

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
// Выбор файла с текстом и его загрузка в текстовое поле
begin
  if OpenFileDialog1.Execute then
    Mem1.Lines.LoadFromFile(OpenDialog1.FileName)
end;

procedure TForm1.Button3Click(Sender: TObject);
// Выбор шрифта и связывание его с текстовым полем
begin
  if FontDialog1.Execute then
    Mem1.Font := FontDialog1.Font
end;

procedure TForm1.Button2Click(Sender: TObject);
// Печать содержимого текстового поля как вывод
// в текстовый файл
var
  Prn: TextFile;
  k: Integer;
begin
  AssignPrn(Prn); // Переназначаем вывод с файла на принтер
  Rewrite(Prn);   // Готовим принтер к печати
                  // (аналог BeginDoc)
  { Для печати используем такой же шрифт, как и для показа
    в текстовом поле: }
  Printer.Canvas.Font := Mem1.Font;
  // Цикл печати:
  for k := 0 to Mem1.Lines.Count-1 do
    WriteLn(Prn, Mem1.Lines[k]);
  CloseFile(Prn); // Аналог EndDoc
end;

end.

```

Такой способ печати — самый примитивный. С его помощью невозможно вывести линии, разделяющие колонки или строки, трудно форматировать текст, вставлять заголовки, номера страниц и т. п.

Значительно более гибкие средства обеспечивает свойство `Printer.Canvas`. В листинге 22.3 показано, как с его помощью можно напечатать текст, содержащийся в поле `Memol` (проект `Chap_22\Print of Text(Canvas)\PrintText.dpr`).

Листинг 22.3. Печать текста с помощью свойства `Printer.Canvas`

```

procedure TForm1.Button2Click(Sender: TObject);
// Печать содержимого поля с помощью свойства
Printer.Canvas
var
    Y,dY,X,k: Integer;
    S: String;
begin
    if Memol.Lines.Count=0 then Exit;
    Screen.Cursor := crHourGlass;
    with Printer do
    begin
        BeginDoc;
        with Canvas do
        begin
            Font := Memol.Font;
            dY := TextHeight('1'); // Определяем высоту строки
            Y := 3*dY;           // Отступ от верхнего края листа
            X := PageWidth div 15; // Отступ от левого края
            for k := 0 to Memol.Lines.Count-1 do
            begin
                // Выводим очередную строку
                TextOut(X,Y,Memol.Lines[k]);
                // Смещаемся на следующую строку листа
                inc(Y,dY);
                if PageHeight-Y<2*dY then // Нижний край листа?
                begin // Да
                    NewPage; // Переход на новый лист
                    // Выводим номер страницы посередине листа:
                    S := '- '+IntToStr(PageNumber)+' -';
                    TextOut((PageWidth-TextWidth(S)) div 2, dy, S);
                    // и отчеркиваем его от текста:
                    MoveTo(X, 3*dy div 2);
                    LineTo(PageWidth-X, 9*dy div 4);
                    // Ордината первой строки:
                    Y := 3*dY
                end; // if PageHeight-Y<2*dY
            end; // for k := 0 to Memol.Lines.Count-1 do
        end; // with Canvas do
        EndDoc;
    end; // with Printer do
    Screen.Cursor := crDefault;
end;

```

Как видим, прямое обращение к графическим инструментам свойства Canvas требует от программиста значительно бóльших усилий, но зато предоставляет ему полный контроль над печатным изображением.

Наконец, очень хороших результатов можно достичь, используя специализированные средства просмотра/печати документов, такие как текстовый процессор MS Word.

Печать изображений может показаться очень сложным делом, однако свойство Printer.Canvas содержит метод, который легко справляется с этой задачей:

```
procedure StretchDraw(const Rect: TRect; Graphic:
    TGraphic );
```

При обращении к этому методу первым параметром указывается прямоугольная область, отводимая на поверхности листа для распечатки изображения, вторым — объект класса TGraphic, в котором хранится изображение. Например:

```
with Printer do
begin
    BeginDoc;
    Canvas.StretchDraw(Canvas.ClipRect,
        Image1.Picture.Graphic);
    EndDoc;
end;
```

Не следует забывать также о том, что любая форма, являющаяся потомком базового класса TCustomForm, получает метод Print, с помощью которого на принтере печатается клиентская часть формы (без заголовка и рамки).

Clipboard

В Windows широко используется буфер обмена (*clipboard*). Если в программе Delphi сослаться на модуль Clipbrd, для нее становится доступным автоматически созданный объект Clipboard, инкапсулирующий свойства (табл. 22.7) и методы (табл. 22.8) для работы с буфером обмена.

Таблица 22.7. Свойства объекта Clipboard

Свойство	Описание
property AsText: String;	Рассматривает содержимое буфера как текстовую строку неограниченной длины
property FormatCount: Integer;	Возвращает общее количество форматов хранящихся в буфере данных
property Formats[Index: Integer]: Word;	Открывает доступ к данным по формату, указанному индексом Index

Таблица 22.8. Методы объекта Clipboard

Метод	Описание
procedure Assign(Source: TPersistent);	Копирует изображение в буфер обмена
procedure Clear;	Очищает буфер обмена

продолжение ➤

Таблица 22.8 (продолжение)

Метод	Описание
procedure Close;	Уменьшает на 1 счетчик блокировок буфера обмена и закрывает его, если счетчик содержит 0
function GetTextBuf (Buffer: PChar; BufSize: Integer): Integer;	Копирует из буфера обмена не более BufSize символов в текстовый буфер Buffer и возвращает действительное количество скопированных символов
procedure HasFormat (Format: Word): Boolean;	Возвращает значение True, если буфер обмена содержит данные в формате Format
procedure Open;	Увеличивает на 1 счетчик блокировок буфера обмена и открывает его, если он еще не был открыт
procedure SetTextBuf (Buffer: PChar);	Помещает в буфер обмена содержимое текстового буфера Buffer

Большая часть компонентов, предназначенных для хранения текста и/или изображений, имеют собственные методы для работы с буфером обмена. Например, текстовые поля TEdit, TMemo, TRichEdit имеют методы CopyToClipboard (Копировать в буфер), CutToClipboard (Вырезать и поместить в буфер), PasteFromClipboard (Вставить из буфера); компонент TImage имеет свойство Picture класса TPicture, которое инкапсулирует методы LoadFromClipboardFormat, SaveToClipboardFormat и RegisterClipboardFormat — с помощью этих методов компонент обменивается с буфером изображениями определенного формата и может зарегистрировать в нем новый формат изображения. Тем не менее эти и другие компоненты для работы с буфером обмена могут использовать свойство AsText объекта Clipboard и его методы Assign, GetTextBuffer, SetTextBuffer. Например, чтобы поместить в буфер обмена изображение, хранящееся в объекте Image1, можно обратиться к методу Assign буфера:

```
Clipboard.Open;
Clipboard.Assign(Image1.Picture);
Clipboard.Close;
```

А для копирования изображения из буфера обмена можно обратиться к методу Picture.Assign компонента Image1:

```
Clipboard.Open;
Image1.Picture.Assign(Clipboard);
Clipboard.Close;
```

Метка не имеет средств работы с буфером обмена, но программа может прочитать в нее текст из буфера:

```
Clipboard.Open;
Label1.Caption := Clipboard.AsText;
Clipboard.Close;
```

Можно также поместить в буфер обмена содержимое свойства Caption метки:

```
Clipboard.Open;
Clipboard.AsText := Label1.Caption;
Clipboard.Close;
```

Настройка программы

Удобным средством хранения индивидуальных параметров программы являются широко используемые в Windows 3.x файлы инициализации (файлы с расширением INI). Для 32-разрядных версий Windows индивидуальная настройка программ чаще всего реализуется с помощью системного реестра. В этом разделе рассматриваются оба способа, так как даже если вы работаете с 32-разрядной версией Windows, перенос программ на другие компьютеры и их настройку проще реализовать с помощью файлов инициализации.

Файлы инициализации

Файлы инициализации в Delphi связаны с объектами класса `TIniFile`. Эти объекты не являются глобальными и создаются программно по мере надобности. С каждой программой можно связать сколько угодно файлов инициализации.

Физически файл инициализации представляет собой обычный текстовый файл, который сохраняется либо в каталоге запуска программы (локальный файл инициализации), либо в каталоге запуска Windows (глобальный файл инициализации). Файл содержит ряд секций, каждая из которых имеет заголовок и следующие за ним параметры. Заголовок — это произвольный английский текст, заключенный в квадратные скобки. Каждый параметр представлен строкой вида:

```
имя_параметра=значение
```

Пример:

```
[Location]
DataBase=C:\MYBASE
Graphics=C:\BITMAP
```

В этом примере в секции `Location` файла инициализации представлены два параметра с именами `DataBase` и `Graphics`. Объекты класса `TIniFile` предоставляют удобные средства чтения/записи параметров по именам параметров и секций.

В объекте имеется единственное свойство `FileName: String`, содержащее имя файла инициализации с возможным путем поиска.

Методы класса `TIniFile` перечислены в табл. 22.9.

Таблица 22.9. Методы класса `TIniFile`

Метод	Описание
constructor <code>Create(const FileName String);</code>	Создаст объект-файл с именем <code>FileName</code> . Это имя автоматически переносится в свойство <code>FileName</code> объекта. Если физический файл существовал, он открывается, если нет — создается

продолжение ➤

Таблица 22.9 (продолжение)

Метод	Описание
procedure DeleteKey(const Section, Ident: String);	Удаляет значение параметра Ident в секции Section. Если указана несуществующая секция или имя несуществующего параметра, метод создаст секцию и/или установит в ней параметр с пустым значением
procedure EraseSection(const Section: String);	Удаляет секцию со всеми ее параметрами. Если секции не существует, возникает исключительная ситуация
function ReadBool (const Section, Ident: String ; Default: Boolean): Boolean;	Возвращает значение логического параметра Ident в секции Section (см. ниже первое примечание)
function ReadDate (const Section, Ident: String ; Default: TDateTime): TDateTime; virtual ;	Возвращает значение параметра Ident типа даты в секции Section (см. ниже первое примечание)
function ReadDateTime (const Section, Ident: String ; Default: TDateTime): TDateTime; virtual ;	Возвращает значение параметра Ident типа даты-времени в секции Section (см. ниже первое примечание)
function ReadFloat (const Section, Ident: String ; Default: Double): Double; virtual ;	Возвращает значение параметра Ident вещественного типа в секции Section (см. ниже первое примечание)
function ReadInteger(const Section, Ident: String ; Default: LongInt): LongInt;	Возвращает значение целого параметра Ident в секции Section (см. ниже первое примечание)
procedure ReadSection (const Section: String ; Strings: TStrings);	Помещает в список Strings имена всех параметров из секции Section
procedure ReadSections(Strings: TStrings);	Помещает в список Strings имена всех секций файла
procedure ReadSectionValues(const Section: String ; Strings: TStrings);	Помещает в список Strings имена всех значений из секции Section
function ReadString(const Section, Ident, Default: String): String ;	Возвращает значение строкового параметра Ident в секции Section (см. ниже первое примечание)
function ReadTime (const Section, Ident: String ; Default: TDateTime): TDateTime; virtual ;	Возвращает значение параметра Ident типа времени в секции Section (см. ниже первое примечание)
function SectionExists (const Section: String): Boolean;	Возвращает значение True, если секция Section существует
function ValueExists (const Section, Ident: String): Boolean;	Возвращает значение True, если у параметра Ident секции Section существует значение

Метод	Описание
procedure WriteBool(const Section, Ident: String ; Value: Boolean);	Записывает в параметр Ident секции Section логическое значение Value (см. ниже второе примечание)
procedure WriteDate(const Section, Ident: String ; Value: TDateTime); virtual ;	Записывает в параметр Ident секции Section значение Value типа даты (см. ниже второе примечание)
procedure WriteDateTime(const Section, Ident: String ; Value: TDateTime); virtual ;	Записывает в параметр Ident секции Section значение Value типа даты-времени (см. ниже второе примечание)
procedure WriteFloat(const Section, Ident: String ; Value: Double); virtual ;	Записывает в параметр Ident секции Section вещественное значение Value (см. ниже второе примечание)
procedure WriteInteger(const Section, Ident: String ; Value: LongInt);	Записывает в параметр Ident секции Section целое значение Value
procedure WriteString(const Section, Ident, Value: String);	Записывает в параметр Ident секции Section строковое значение Value
procedure WriteTime(const Section, Ident: String ; Value: TDateTime); virtual ;	Записывает в параметр Ident секции Section значение Value типа времени (см. ниже второе примечание)

ПРИМЕЧАНИЕ В функциях ReadXXXX (ReadBool, ReadData и т. п.) используется параметр Default того же типа, что и тип возвращаемого результата. Если функция не найдет указанной секции или нужного параметра, а также если значение параметра окажется не ожидаемого типа, она возвращает значение Default.

ПРИМЕЧАНИЕ Если при обращении к процедуре WriteXXXX (WriteBool, WriteData и т. п.) не существует указанной секции и/или параметра, они будут созданы.

В листинге 22.4 представлен модуль, иллюстрирующий приемы работы с файлом инициализации, а на рис. 22.1 показано создаваемое им окно (проект Chap_22\IniFile\IniFile.dpr).

Листинг 22.4. Иллюстрация работы с файлом инициализации

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Spin;
```

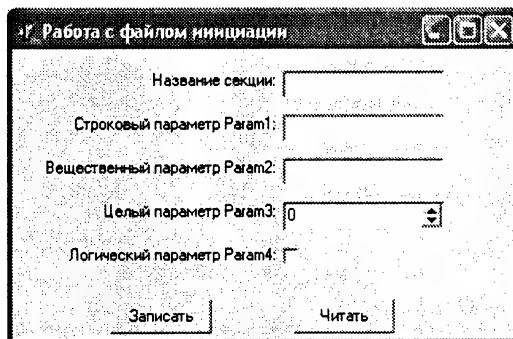


Рис. 22.1. Иллюстрация работы с файлом инициализации

type

```

TForm1 = class (TForm)
  Label1: TLabel;
  Label2: TLabel;
  Label3: TLabel;
  Label4: TLabel;
  Label5: TLabel;
  edSection: TEdit;
  edParam1: TEdit;
  edParam2: TEdit;
  cbParam4: TCheckBox;
  Button1: TButton;
  Button2: TButton;
  seParam3: TSpinEdit;
  procedure Button1Click(Sender: TObject);
  procedure Button2Click(Sender: TObject);
  procedure edParam2Exit(Sender: TObject);
private
  { Private declarations }
public
  { Public declarations }
end;

```

var

```
Form1: TForm1;
```

implementation

```

uses IniFiles; // Обязательная ссылка
               // для доступа к типу TIniFile
const
  SectionName = 'InitSection'; // Название секции по
                               // умолчанию

```

```

    IniFileName = 'DemoIni.ini';  // Имя файла инициализации
var
    Ini: TIniFile;

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
// Создает или изменяет файл инициализации
begin
    if edSection.Text='' then
        edSection.Text := SectionName;
    Ini := TIniFile.Create(IniFileName);
    Ini.WriteString(edSection.Text, 'Param1', edParam1.Text);
    Ini.WriteFloat(edSection.Text, 'Param2',
        StrToFloat (edParam2.Text));
    Ini.WriteInteger(edSection.Text, 'Param3',
        seParam22.Value);
    Ini.WriteBool(edSection.Text, 'Param4',
        cbParam4.Checked);
    Ini.Destroy;
end;

procedure TForm1.Button2Click(Sender: TObject);
// Читает параметры настройки
begin
    if edSection.Text='' then
        edSection.Text := SectionName;
    Ini := TIniFile.Create(IniFileName);
    edParam1.Text := Ini.ReadString(edSection.Text,
        'Param1', 'Delphi');
    edParam2.Text := FloatToStr(Ini.ReadFloat(edSection.Text,
        'Param2', 22.1415));
    seParam22.Value := Ini.ReadInteger(edSection.Text,
        'Param3', 100);
    cbParam4.Checked := Ini.ReadBool(edSection.Text,
        'Param4', True);
    Ini.Destroy;
end;

procedure TForm1.edParam2Exit(Sender: TObject);
// Проверяет правильность символьного представления
// вещественного числа
begin
    try
        StrToFloat(edParam2.Text)

```

```

except
    edParam2.SelectAll;
    edParam2.SetFocus;
end
end;

end.

```

Отмечу, что если путь доступа к файлу инициализации не указан (как в нашем примере), считается, что он располагается в системном каталоге (обычно C:\Windows).

Системный реестр Windows

Системный реестр Windows — это общедоступная база данных, хранящая индивидуальную для каждого компьютера настроечную информацию, используемую программным обеспечением. Данные в реестре упорядочены в древовидные структуры, состоящие из узлов, которые в терминологии Windows называются ключами. Каждый ключ имеет один родительский ключ, один или несколько дочерних ключей и ноль или несколько пар типа имя_параметра=значение. Исключение представляют корневые ключи (они не имеют родителей) и дочерние ключи низшего уровня (они не имеют дочерних ключей). По умолчанию несистемное программное обеспечение регистрирует свою информацию в корневом ключе с именем HKEY_CURRENT_USER. Для просмотра/коррекции системного реестра используется утилита REGEDIT.EXE.

Системному реестру Windows соответствует объект класса TRegistry, свойства которого представлены в табл. 22.10.

Таблица 22.10. Свойства класса TRegistry

Свойство	Описание
property Access: LongWord;	Определяет уровень доступа к открытому ключу. Значение свойства может представлять собой некоторую комбинацию флагов (см. ниже)
property CurrentKey: HKEY;	Содержит текущий раскрытый ключ
property CurrentPath: String;	Содержит путь поиска для текущего ключа
property LazyWrite: Boolean;	Если содержит значение True, ключ записывается при выполнении процедуры CloseKey, в противном случае — при любом изменении ключа
property RootKey: HKEY;	Содержит имя корневого ключа

Перечень флагов для свойства Access:

- KEY_QUERY_VALUE — разрешает запрос значений дочерних ключей;
- KEY_ENUMERATE_SUB_KEYS — разрешает получение списка всех дочерних ключей;

- KEY_NOTIFY — разрешает получение извещения об изменении ключа;
 - KEY_SET_VALUE — разрешает изменение значений дочерних ключей;
 - KEY_CREATE_SUB_KEY — разрешает создание дочерних ключей;
 - KEY_CREATE_LINK — разрешает создание строковой ссылки;
 - KEY_EXECUTE — определяет доступ к ключу в режиме «только для чтения».
 - KEY_ALL_ACCESS — комбинация флагов KEY_READ, KEY_WRITE и KEY_CREATE_LINK;
 - KEY_READ — комбинация флагов KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS и KEY_NOTIFY;
 - KEY_WRITE — комбинация флагов KEY_SET_VALUE и KEY_CREATE_SUB_KEY,
- Методы класса TRegistry перечислены в табл. 22.11.

Таблица 22.11. Методы класса TRegistry

Метод	Описание
procedure CloseKey;	Записывает информацию в ключ и закрывает его
constructor Create;	Создает объект класса TRegistry. Устанавливает в RootKey значение HKEY_CURRENT_USER и в LazyWrite — значение True
function CreateKey(const Key: String): Boolean;	Создает ключ с именем Key. Если имя начинается символом \, ключ является дочерним ключом корневого ключа, в противном случае — дочерним ключом текущего ключа
function DeleteKey(const Key: String): Boolean;	Удаляет ключ Key и возвращает значение True, если операция прошла успешно
function DeleteValue(const Name: String): Boolean;	В текущем узле удаляет значение параметра с именем Name
destructor Destroy;	Удаляет объект TRegistry
TRegDataType = (rdUnknown, rdString, rdExpandString, rdInteger, rdBinary); TRegDataInfo = record RegData: TRegDataType; DataSize: Integer; end ; function GetDataInfo(const ValueName: String; var Value: TRegDataInfo): Boolean;	Возвращает в параметре Value значение, определяющее тип данных для параметра с именем ValueName текущего ключа: rdUnknown — неизвестный тип; rdString — строковый тип; rdExpandString — значением является строка, описывающая переменную окружения Windows, например "PATH%"; rdInteger — целочисленный тип; rdBinary — двоичный тип (набор битов)
function GetDataSize(const ValueName: String): Integer;	Возвращает длину значения параметра с именем ValueName в текущем узле
TRegDataType = (rdUnknown, rdString, rdExpandString, rdInteger, rdBinary);	Возвращает значение, определяющее тип данных для параметра с именем ValueName текущего ключа (см. метод

продолжение ➤

Таблица 22.11 (продолжение)

Метод	Описание
function GetDataType(const ValueName: String): TRegDataType;	GetTypeInfo()
TRegKeyInfo = record NumSubKeys: Integer; MaxSubKeyLen: Integer; NumValues: Integer; MaxValueLen: Integer; MaxDataLen: Integer; FileTime: TFileTime; end ; function GetKeyInfo(var Value: TRegKeyInfo): Boolean;	Возвращает в параметре Value информацию о текущем узле: NumSubKeys — количество дочерних узлов; MaxSubKeyLen — максимальная длина имени дочернего узла; NumValues — количество параметров; MaxValueLen — максимальная длина имени параметра; MaxDataLen — максимальная длина параметра; FileTime — время последнего обновления ключа
procedure GetKeyNames (Strings: TStrings);	Возвращает в параметре Strings имена всех дочерних ключей
procedure GetValueNames (Strings: TStrings);	Возвращает в параметре Strings имена всех параметров
function HasSubKeys: Boolean;	Возвращает значение True, если ключ имеет дочерние ключи
function KeyExists(const Key: String): Boolean;	Возвращает значение True, если существует ключ с именем Key
function LoadKey(const Key, FileName: String): Boolean;	Создает ключ с именем Key как дочерний ключ корневого ключа и загружает из файла FileName его содержимое (параметры и все дочерние ключи)
procedure MoveKey(const OldName, NewName: String ; Delete: Boolean);	Копирует или перемещает ключ OldName в ключ NewName: Delete — содержит значение True для перемещения
function OpenKey(const Key: String ; CanCreate: Boolean): Boolean;	Открывает существующий или создаст и открывает новый ключ с именем Key и возвращает значение True, если операция прошла успешно. Параметр CanCreate разрешает/запрещает создание ключа, если он не существует
function OpenKeyReadOnly (const Key: String): Boolean;	Открывает ключ в режиме «только для чтения»
Function ReadBinaryData (const Name: String ; var Buffer; BufSize: Integer): Integer;	Копирует не более BufSize байтов в переменную Buffer из параметра с именем Name. Возвращает количество действительно скопированных байтов
Function ReadXXXX(const Name: String): YYYY;	Возвращает значение параметра типа YYYY с именем Name

Метод	Описание
Function RegistryConnect (const UNCName: String): Boolean;	Разрешает удаленному компьютеру UNCName доступ к системному реестру вашего компьютера
Procedure RenameValue (const OldName, NewName: String);	Переименовывает параметр OldName
Function ReplaceKey(const Key, FileName, BackUpFileName: String): Boolean;	Заменяет содержимое ключа Key содержимым файла FileName. BackUpFileName — имя файла, в котором будет сохраняться прежнее содержимое ключа
Function RestoreKey (const Key, FileName: String): Boolean;	Считывает содержимое ключа Key из файла FileName
Function SaveKey(const Key, FileName: String): Boolean;	Сохраняет содержимое ключа Key в файле FileName
Function ValueExists(const Name: String): Boolean;	Возвращает значение True, если параметр с именем Name имеет значение
Procedure WriteBinaryData (const Name: String ; var Buffer; BufSize: Integer);	Копирует не более BufSize байтов из буфера Buffer в параметр с именем Name
Procedure WriteXXXX(const Name: String ; Value: YYYY);	Записывает в параметр с именем Name данные типа YYYY

Системный реестр — удобное место для размещения некоторых «секретных» данных, которые не должны быть на виду у пользователей. Например, при первом запуске пробных версий программ в него помещается дата запуска или максимальное количество запусков. Перед очередным прогоном эта информация проверяется, в результате после некоторого количества запусков или по истечении установленного времени пробная версия программы перестает нормально работать.

В листинге 22.5 представлен файл проекта, в котором пробная версия программы может запускаться не более 5 раз (см. проект Chap_22\Registry\Trial.dpr).

Листинг 22.5. Создание пробной версии программы

```

program Trial;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Registry, Dialogs; // Для TRegistry и ShowMessage

{$R *.RES}
var
  Reg: TRegistry;
```



```

N: Integer;
begin
  Reg := TRegistry.Create;
  with Reg do
    begin
      OpenKey('software', True);
      OpenKey('TrialProg', True);
      if ValueExists('MaxRun') then // Первый запуск?
        begin // - Нет
          N := ReadInteger('MaxRun')-1;
          if N>=0 then
            WriteInteger('MaxRun', N)
          end else begin // - Да, первый запуск
            N := 5;
            WriteInteger('MaxRun', N)
          end;
        Free
      end;
      if N>0 then
        begin
          Application.CreateForm(TForm1, Form1);
          Form1.Label2.Caption := IntToStr(N-1);
          Application.Run;
        end else
          ShowMessage('Исчерпано максимальное количество
            запусков '+' пробной версии программы')
        end.
    end.
  end.

```

Небольшой комментарий. Двумя обращениями к функции `TRegistry.OpenKey` создается и/или открывается узел `HKEY_CURRENT_USER\Software\TrialProg` системного реестра. Функция `TRegistry.ValueExists` возвращает значение `True`, если в этом узле есть параметр с именем `MaxRun` и для него определено значение. При первом запуске программы это не так, поэтому процедурой `WriteInteger` создается параметр и для него указывается начальное значение 5 (максимальное количество прогонов программы). При каждом следующем запуске этот параметр уменьшается на 1, и в момент, когда он становится равным нулю, программа блокирует создание и вызов главного окна.

Delphi создает надстройку над классом `TRegistry` для того, чтобы манипуляция с системным реестром Windows была подобна манипуляциям с файлами инициализации. Эта надстройка инкапсулируется в классе `TRegIniFile`.

Свойства класса `TRegIniFile` аналогичны свойствам класса `TRegistry` за исключением дополнительного свойства `FileName: String`, хранящего имя ключа, для которого будут осуществляться нужные действия. Эти действия реализуются методами класса, перечисленными в табл. 22.12.

Таблица 22.12. Методы класса TRegIniFile

Метод	Описание
constructor Create(const FileName: String);	Создает объект класса TRegIniFile и связывает его с ключом FileName
procedure DeleteKey(const Section, Ident: String);	Удаляет параметр с именем Ident в ключе Section
procedure EraseSection(const Section: String);	Удаляет ключ Section со всеми его параметрами и дочерними ключами
function ReadBool(const Section, Ident: String ; Default: Boolean): Boolean;	Возвращает значение логического параметра с именем Ident в ключе Section
function ReadInteger(const Section, Ident: String ; Default: LongInt): LongInt;	Возвращает значение целочисленного параметра с именем Ident в ключе Section
procedure ReadSection(const Section: String ; Strings: TStrings);	Возвращает в параметре Strings имена всех параметров ключа Section
procedure ReadSections(Strings: TStrings);	Возвращает в параметре Strings имена всех ключей, связанных с текущим ключом
procedure ReadSectionValues(const Section: String ; Strings: TStrings);	Возвращает в параметре Strings значения всех параметров ключа Section
function ReadString(const Section, Ident, Default: String): String ;	Возвращает значение строкового параметра с именем Ident в ключе Section
procedure WriteBool(const Section, Ident: String ; Value: Boolean);	Записывает логическое значение Value в параметр Ident ключа Section
procedure WriteInteger(const Section, Ident: String ; Value: LongInt);	Записывает целочисленное значение Value в параметр Ident ключа Section
procedure WriteString(const Section, Ident, Value: String);	Записывает строковое значение Value в параметр Ident ключа Section

ПРИМЕЧАНИЕ В ключе HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run хранится перечень программ, которые будут автоматически запускаться в момент старта Windows.

Программа и Windows

Delphi — это всего лишь одно из многочисленных приложений, написанных в расчете на работу под управлением операционной системы Windows. В этом разделе приводятся минимально необходимые сведения о Windows, позволяющие понять и грамотно использовать соответствующие средства Delphi.

Процессы и потоки

Процесс — это специальный объект Windows, который создается для каждой исполняющейся под управлением этой операционной системы программы. Сам по себе процесс ничего не делает, просто в нем инкапсулируется вся необходимая информация о программе. Чтобы программа могла выполняться, помимо процесса создается еще поток команд, который называется первичным или главным. В программе можно создать сколько угодно потоков команд, однако в каждый момент времени активен (то есть использует ресурсы процессора) только один из них. Потоки получают доступ к процессору небольшими квантами времени, поэтому у пользователя складывается впечатление, что все потоки команд работают параллельно.

Один процесс может породить другой процесс. Это означает, что одна работающая программа может инициировать выполнение другой программы. В Windows могут параллельно исполняться множество программ, то есть эта операционная система обладает свойствами многозадачности и многопоточности. Создать процесс можно с помощью функции `CreateProcess`:

```
function CreateProcess(lpApplicationName: PChar;
  lpCommandLine: PChar;
  lpProcessAttributes, lpThreadAttributes:
  PSecurityAttributes; bInheritHandles: BOOL;
  dwCreationFlags: DWORD; lpEnvironment: Pointer;
  lpCurrentDirectory: PChar;
  const lpStartupInfo: TStartupInfo;
  var lpProcessInformation: TProcessInformation): BOOL;
```

Назначение параметров описано ниже.

- ✿ `lpApplicationName` — имя исполняемого файла с возможным путем доступа. Если путь доступа опущен, имя дополняется текущим диском и текущим каталогом. Если параметр имеет значение **NIL**, имя исполняемого файла должно указываться первым в перечне параметров командной строки `lpCommandLine` и отделяться от других параметров пробелом. Если запускается 16-разрядная программа под управлением Windows NT/2000/XP, параметр `lpApplicationName` должен быть равен **NIL**.
- ✿ `lpCommandLine` — строка командных параметров. Если параметр имеет значение **NIL**, командные параметры можно передавать программе в параметре `lpApplicationName`. Если запускается 16-разрядная программа под управлением Windows NT/2000/XP, первым в списке параметров должно стоять имя исполняемого файла с возможным путем доступа.

- `lpProcessAttributes` — указатель на структуру, описывающую привилегии доступа процесса. Имеет смысл только для Windows NT/2000/XP. Если имеет значение **NIL**, процессу присваиваются привилегии доступа, заданные по умолчанию.
- `lpThreadAttributes` — указатель на структуру, описывающую привилегии доступа главного потока команд программы. Имеет смысл только для Windows NT/2000/XP. Если имеет значение **NIL**, потоку присваиваются привилегии доступа, заданные по умолчанию.
- `bInheritHandles` — указывает, будет ли новый процесс наследовать дескрипторы вызывающего процесса. Если имеет значение `True`, порождаемый процесс разделяет память, карту файлов и другие системные атрибуты со своим родителем.
- `dwCreationFlags` — один или несколько флагов, уточняющих способ создания нового процесса и его приоритет:
 - ☐ `CREATE_DEFAULT_ERROR_MODE` — новый процесс получает свой обработчик ошибок;
 - ☐ `CREATE_NEW_CONSOLE` — новый процесс получает свои консольные средства ввода-вывода; не совместим с флагом `DETACHED_PROCESS`;
 - ☐ `CREATE_NEW_PROCESS_GROUP` — новый процесс будет корневым для группы процессов;
 - ☐ `CREATE_SEPARATE_WOW_VDM` — новый 16-разрядный процесс Windows будет запущен в отдельной виртуальной машине DOS; имеет смысл только для Windows NT/2000/XP;
 - ☐ `CREATE_SHARED_WOW_VDM` — проверяет параметр `DefaultSeparateVDM` в файле `Win.ini` и, если он имеет значение `True`, запускает новый 16-разрядный процесс в отдельной виртуальной машине DOS, в противном случае — в общей виртуальной машине; имеет смысл только для Windows NT/2000/XP;
 - ☐ `CREATE_SUSPENDED` — после создания нового процесса его главный поток команд не будет активным (его можно впоследствии активизировать обращением к функции `ResumeThread`);
 - ☐ `CREATE_UNICODE_ENVIRONMENT` — если указан, блок заканчивающихся нулем строк `Environment` имеет кодировку Unicode;
 - ☐ `DEBUG_PROCESS` — новый процесс будет отлаживаться вызывающим процессом;
 - ☐ `DEBUG_ONLY_THIS_PROCESS` — если не установлен, каждый процесс, порождаемый новым отлаживаемым процессом, становится еще одним отлаживаемым процессом вызывающего процесса;
 - ☐ `DETACHED_PROCESS` — если установлен, новый консольный процесс не имеет собственных консольных средств ввода-вывода, но может их запросить обращением к функции `AllocConsole`; не совместим с флагом `CREATE_NEW_CONSOLE`;

- ❑ `HIGH_PRIORITY_CLASS` — присваивает главному потоку команд нового процесса высокий приоритет;
- ❑ `IDLE_PRIORITY_CLASS` — главный поток нового процесса будет активизироваться только тогда, когда нет ни одного активного потока;
- ❑ `NORMAL_PRIORITY_CLASS` — присваивает обычный приоритет;
- ❑ `REALTIME_PRIORITY_CLASS` — присваивает наивысший приоритет.
- `lpEnvironment` — указатель на оканчивающийся нулем блок оканчивающихся нулем строк, описывающих переменные окружения нового процесса (например, каталоги поиска по умолчанию). Каждая переменная задается строкой вида параметр=значение. Если содержит **NIL**, процесс использует значения переменных окружения, заданные по умолчанию.
- `lpCurrentDirectory` — указатель на оканчивающуюся нулем строку, устанавливающую для нового процесса диск и каталог, заданные по умолчанию.
- `lpStartupInfo` — указатель на структуру, описывающую положение и размеры главного окна нового процесса, его средства ввода-вывода и т. п.
- `lpProcessInformation` — указатель на структуру, создаваемую функцией `CreateProcess`, в которой содержится информация о вновь созданном процессе и его главном потоке команд.

Если вызов был успешным, функция возвращает значение `True`.

В листинге 22.6 показан пример вызова функции `CreateProcess`. В каталоге запуска примера должен располагаться файл `Child.exe` (см. каталог `Chap_22\CreateProcess` среди файлов к книге).

Листинг 22.6. Пример вызова функции `CreateProcess`

```
procedure TForm1.Button1Click(Sender: TObject);
// Пример запуска дочернего процесса
var
    StartupInfo: TStartupInfo;
    ProcInfo: TProcessInformation;
const
    CommandLine = 'Child.exe';
begin
    { Готовим структуру StartupInfo. Подробности о назначении
    ее полей см. в справочной службе Win32 SDK Reference}
    FillChar(StartupInfo, SizeOf(StartupInfo), 0);
    with StartupInfo do
    begin
        cb := SizeOf(StartupInfo);    // Указываем размер
                                     // структуры
        { Флаг STARTF_USESHOWWINDOW заставляет учитывать параметр
        wShowWindow. Флаг STARTF_FORCEONFEEDBACK переводит
        указатель мыши в режим "обратной связи" — он ждет окончания
        создания дочернего процесса. }
        dwFlags := STARTF_USESHOWWINDOW or
            STARTF_FORCEONFEEDBACK;
```

```

// Окно нового процесса должно быть видимым
wShowWindow := sw_ShowNormal;
end;
{ Создаем дочерний процесс. В общем случае указывать имя
исполняемого файла в параметре CommandLine разумнее, так
как обеспечивает правильный вызов 16-разрядных приложений
в Windows NT }
if not CreateProcess(NIL, PChar(CommandLine), NIL, NIL,
False, NORMAL_PRIORITY_CLASS, NIL, NIL, StartUpInfo,
ProcInfo) then
    ShowMessage('Ошибка '+IntToStr(GetLastError))
end;

```

В целях совместимости с 16-разрядной версией Windows, в Win32 API включена следующая функция:

function WinExec(lpCmdLine: LPCSTR; uCmdShow: UINT): UINT;

Эта функция имеет более простой интерфейс:

- lpCmdLine — указатель на командную строку запуска (имя исполняемого файла и передаваемые программе параметры запуска);
- uCmdShow — параметр, определяющий состояние главного окна нового процесса:
 - SW_HIDE — главное окно невидимо;
 - SW_MAXIMIZE, SW_SHOWMAXIMIZED — окно имеет максимальные размеры;
 - SW_MINIMIZE, SW_SHOWMINIMIZED — окно свернуто до кнопки;
 - SW_RESTORE, SW_SHOW, SW_SHOWDEFAULT, SW_SHOWNA, SW_SHOWNORMAL — окно нормальных размеров и занимает стандартную позицию на экране;
 - SW_SHOWNOACTIVATE — окно появляется на экране, но не становится активным.

Если вызов был успешным, функция возвращает значение больше 31.

Современные 32-разрядные операционные системы Windows обеспечивают не только многозадачность, то есть возможность параллельной работы нескольких программ, но и многопоточность, когда в рамках одной программы организуется несколько параллельно выполняемых фрагментов (потоків команд), каждый из которых конкурирует с другими потоками за наиболее важный ресурс — время центрального процессора. В многопоточном режиме время центрального процессора выделяется для каждого потока команд небольшими порциями (квантами), по истечении этого времени управление передается другому потоку и т. д. до тех пор, пока потоки не закончат свою работу. В любой работающей программе организуется как минимум один поток для команд программы. С помощью объектов класса TThread программа может создать дополнительные потоки для проведения некоторой фоновой работы (например, текстовый процессор Word создает дополнительные потоки для проверки орфографии, разбивки на страницы, печати документа и т. п.).



Thread Object

Для создания дополнительного потока команд в программах Delphi предназначен специальный модуль потока (значок, показанный рядом с этим абзацем, идентифицирует модуль потока команд в хранилище объектов). При выборе этого модуля Delphi запрашивает имя класса, который будет дочерним для родительского класса TThread. Необходимость наследования связана с тем, что класс TThread содержит абстрактный метод Execute, который, собственно, и должен исполняться в рамках нового потока команд и который, следовательно, обязан перекрываться в потомках.

После указания имени дочернего класса Delphi раскрывает дополнительный модуль с обширным комментарием и заготовкой для дочернего класса. Вот его текст (с соответствующим переводом):

```
unit Unit1;
```

```
interface
```

```
uses
```

```
Classes;
```

```
type
```

```
MyThread = class (TThread)
```

```
private
```

```
{ Private declarations }
```

```
protected
```

```
procedure Execute; override;
```

```
end;
```

```
implementation
```

{ Важно: Методы и свойства объектов из библиотеки визуальных компонентов могут использоваться только в рамках вызова метода Synchronize, например:

```
Synchronize (UpdateCaption);
```

Здесь метод UpdateCaption должен выглядеть примерно так:

```
procedure MyThread.UpdateCaption;
```

```
begin
```

```
Form1.Caption := 'Новый текст метки';
```

```
end; }
```

```
{ MyThread }
```

```
procedure MyThread.Execute;
```

```
begin
```

```
{ Поместите код потока в этом месте }
```

```
end;
```

```
end.
```

Программирование потока команд ничем не отличается от программирования обычной программы за одним важным исключением: поток не должен использовать методы и свойства визуальных компонентов, которые приводят к изменению внешнего вида программы. Точнее, он может это делать только при обращении к специальному методу `Synchronize`, с помощью которого осуществляется синхронизация исполнения главного потока команд программы с дополнительным потоком команд.

Для иллюстрации приемов работы с потоками команд создадим программу, которая будет непрерывно обновлять содержимое многострочного текстового поля и при этом осуществлять математические вычисления. На рис. 22.2 показано окно работающей программы (проект `Chap_22\Thread\Thread.dpr`).

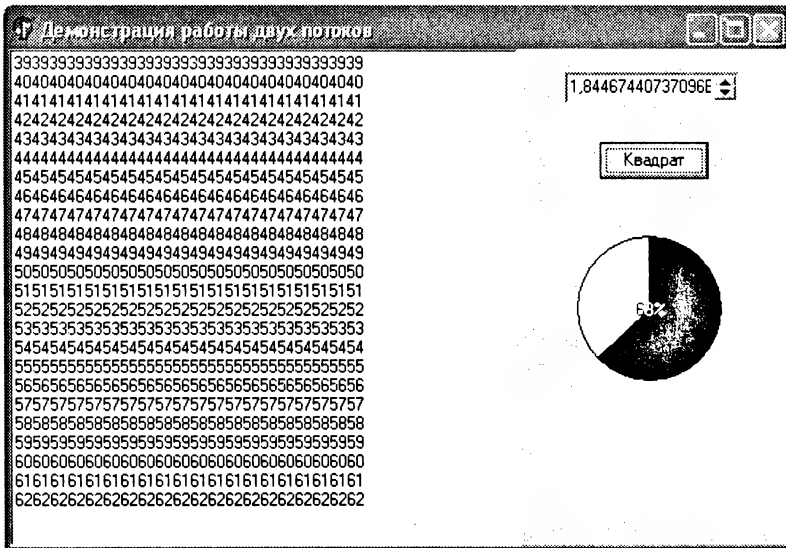


Рис. 22.2. Иллюстрация работы с потоками команд

Основной поток команд программы активизируется при щелчке на кнопке **Квадрат**: вначале содержимое расположенного над ней поля со счетчиком `SpinEdit1` возводится в квадрат до тех пор, пока отображаемое в нем значение не станет слишком большим (больше 10^{1233}). В этот момент надпись на кнопке меняется на **Корень**, а щелчок на ней вычисляет корень квадратный из величины в поле `SpinEdit1`.

Вот обработчик щелчка на этой кнопке:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Tag=0 then
  begin
    SpinEdit1.Text :=
      FloatToStr(sqr(StrToFloat(SpinEdit1.Text)));
    if StrToFloat(SpinEdit1.Text) > 1e1233 then
    begin
      Tag := 1;
```



```

        Button1.Caption := 'Корень'
    end
end else begin
    SpinEdit1.Text :=
        FloatToStr(sqrt(StrToFloat(SpinEdit1.Text)));
    if StrToFloat(SpinEdit1.Text) < 2 then
    begin
        SpinEdit1.Value := 2;
        Tag := 0;
        Button1.Caption := 'Квадрат'
    end
end
end;

```

В дополнительном потоке команд будем непрерывно формировать по 100 строк в многострочном текстовом поле `mmOutput` и показывать процент заполнения поля с помощью индикатора `Gauge1`.

Окончательный текст модуля потока команд представлен ниже.

```

unit Unit2;

interface

uses
    Classes;

type
    ThreadDemo = class(TThread)
    private
        { Private declarations }
    protected
        S: String;
        N: Integer;
        procedure UpdateMemo;
        procedure UpdateGauge;
        procedure Execute; override;
    end;

var
    TDemo: ThreadDemo;

implementation

uses Unit1, SysUtils;

```

{ Important: Methods and properties of objects in VCL can only be used in a method called using Synchronize, for example,

```
Synchronize(UpdateCaption);
```

and UpdateCaption could look like,

```
procedure ThreadDemo.UpdateCaption;  
begin  
  Form1.Caption := 'Updated in a thread';  
end; }
```

```
{ ThreadDemo }
```

```
procedure ThreadDemo.Execute;  
var  
  j,k: Integer;  
begin  
  repeat  
    S := '';  
    Synchronize(UpdateMemo);  
    for k := 0 to 99 do  
      begin  
        N := k;  
        S := '';  
        for j := 1 to 20 do  
          S := S+FormatFloat('00',k);  
          Synchronize(UpdateMemo);  
          Synchronize(UpdateGauge)  
        end;  
      until False  
    end;  
end;
```

```
procedure ThreadDemo.UpdateMemo;  
begin  
  with Form1.mmOutput.Lines do  
    if (S='') or (Count>1000) then  
      Clear  
    else  
      Add(S)  
    end;  
end;
```

```
procedure ThreadDemo.UpdateGauge;  
begin  
  Form1.Gauge1.Progress := N  
end;  
  
end.
```

Дополнительный поток команд запускается в обработчике OnActivate главной формы:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
    TDemo := ThreadDemo.Create(False);
end;
```

Вот так просто запускается дополнительный поток команд — мы инициализируем объект TDemo, передавая в его конструктор ThreadDemo.Create единственный параметр False (этот параметр показывает, должен ли вновь созданный поток команд «спать» — True или он обязан немедленно начать работу — False). Программа в любой момент может приостановить работу потока команд, присвоив его свойству Suspended значение True, и продолжить его выполнение, присвоив этому свойству значение False. Обратите внимание: метод Execute потока вынесен в секцию **protected** и поэтому недоступен из основного модуля. Выполнение этого метода начинается автоматически, как только свойство Suspended примет значение False.

Для обращения к свойствам и методам визуальных компонентов формы Form1 предназначен специальный метод Synchronize потока команд. Единственным параметром обращения к этому методу должно быть имя любой потоковой процедуры *без параметров*. Внутри такой процедуры разрешается обращаться к методам и свойствам визуальных компонентов. В нашем потоке команд имеется две таких процедуры — UpdateMemo и UpdateGauge. В первой строка S добавляется к содержимому поля mmOutput, причем для предотвращения переполнения внутреннего буфера поля оно периодически очищается. В процедуре UpdateGauge глобальная переменная N присваивается свойству Progress индикатора Gauge1. Поскольку эти процедуры не могут иметь параметров, для управления их работой приходится использовать глобальные переменные S и N.

Использование памяти

В 32-разрядных версиях Windows используется так называемая «плоская» модель памяти. Это означает, что каждому процессу выделяется адресное пространство размером 4 Гбайт, причем 32-разрядный указатель определяет порядковый номер ячейки этой воображаемой (виртуальной) памяти. Я не сомневаюсь, что ваш компьютер (как, впрочем, и любой другой «нормальный» компьютер — эти строки пишутся в конце 2002 года) не имеет такого большого объема физической оперативной памяти. Более того, сейчас трудно вообразить программу, которой такая память будет *принципиально* необходима. Речь идет о том, что любой процесс (точнее, любой поток команд любого процесса) *может* обращаться к памяти такого размера. Фактически это означает, что память вашего компьютера объемом 16, 32 и более мегабайт, отображается на гигантское пространство 4 Гбайт. В рамках механизма отображаемой памяти создается так называемый файл подкачки, использующийся для хранения страниц виртуальной памяти, которые превышают размеры физической памяти. Когда программа (поток команд) обращается к адресам, которых нет в физической памяти, на свободное место оперативной памяти считывается фрагмент файла подкачки, а после использования требуемых адресов, он

вновь переписывается на диск. Обмен виртуальной памяти с физической памятью (подкачка) выполняется страницами длиной по 4 Кбайт. Подкачка, разумеется, замедляет скорость работы программы, но зато программист может не заботиться об экономии физической памяти. Замечу, что в 16-разрядной версии Windows использовалась так называемая сегментная модель памяти, в которой память программе выделялась блоками размером по 64 Кбайт. Это создавало определенные ограничения. В частности, ни один массив, запись или любой другой тип произвольного размера не мог превышать длину сегмента — 64 Кбайт. В 32-разрядных версиях Windows это ограничение «отодвинуто» до 4 Гбайт.

ПРИМЕЧАНИЕ Следует уточнить, что первые («верхние») 2 Гбайта виртуальной памяти отводятся ядру операционной системы и недоступны программе.

Плоская модель памяти гарантирует, что *каждый* процесс получит в свое распоряжение отдельный участок виртуальной памяти размером в 4 Гбайт. Это означает, что процессы «не видят» друг друга и не могут общаться через общую память (речь не идет о процессах, созданных с параметром `InheritHandles=True`).

Сообщения Windows

Сообщениями в Windows называются небольшие пакеты информации, которыми обмениваются ядро Windows с программами и программы друг с другом. Сообщение связано с некоторым событием и извещает адресата о его наступлении.

Структура сообщений

В модуле Windows сообщение описано следующим образом:

type

TMsg = packed record

hWnd: HWND; // Дескриптор окна-получателя

Message: UINT; // Код сообщения

WParam: WPARAM; // Уточняющий параметр

LParam: LPARAM; // Уточняющий параметр

Time: DWORD; // Время создания сообщения

pt: TPoint; // Координаты указателя мыши в этот момент

end;

Типы HWND, UINT, DWORD описаны в том же модуле и соответствуют типу LongWord, а типы WPARAM, LPARAM — типу LongInt.

Важно заметить, что сообщение всегда адресуется некоторому окну-получателю, определяемому полем hWnd. Любой оконный класс Delphi порожден от класса TControl и имеет метод WndProc, предназначенный для получения и обработки сообщений (этот метод называется *оконной процедурой*).

Система сообщений

Система сообщений функционирует следующим образом. При создании любого окна программы операционная система Windows выделяет ему некоторые ресурсы и регистрирует его оконную процедуру. Оконная процедура главного окна бу-

дет создана и зарегистрирована путем вызова метода `Application.CreateForm` в файле проекта. При этом Windows выделит дополнительные ресурсы для организации очереди сообщений, адресованных любому окну данной программы. Далее в этом файле осуществляется вызов метода `Application.Run`, ядро которого составляет следующий цикл:

```
repeat
    HandleMessage
until Terminated;
```

В ходе реализации `Application.HandleMessage` вызывается такая последовательность операторов:

```
if PeekMessage(Msg, 0, 0, 0, PM_REMOVE) then
begin
    ...
    TranslateMessage(Msg);
    DispatchMessage(Msg);
    ...
end;
```

Вызывая API-функцию `PeekMessage`, программа извлекает из очереди сообщений новое сообщение, а вызывая методы `TranslateMessage` и `DispatchMessage`, обрабатывает его (преобразует нажатие клавиш в сообщения `WM_CHAR`) и направляет нужной оконной процедуре.

Обработка сообщений

Операционная система Windows построена таким образом, что в подавляющем большинстве случаев программа, которой посылается сообщение, *обязана* должным образом отреагировать на него. Хочу подчеркнуть, что многие сообщения приводят к возникновению событий компонентов. Программа может игнорировать эти события, но обязана обработать сообщение, их породившее.

Значительную часть работы по обработке сообщений программы, созданные с помощью Delphi, выполняют автоматически. Для этого Delphi, в частности, преобразует тип `TMsg` в собственный тип `TMessage`:

```
type
    TMessage = packed record
        Msg: Cardinal;
        case Integer of
            0: (
                WParam: Longint;
                LParam: Longint;
                Result: Longint);
            1: (
                WParamLo: Word;
                WParamHi: Word;
                LParamLo: Word;
                LParamHi: Word;
                ResultLo: Word;
```

```

        ResultHi: Word);
    end;

```

Нетрудно заметить, что запись TMessage содержит меньше информации, чем TMsg; с другой стороны, в нее введено поле Result. Связано это с тем, что программа анализирует характер полученного сообщения и создает многочисленные специализированные сообщения. Например, при нажатии клавиши или ее отпуске создается сообщение такой структуры:

```

type
    TWMKey = packed record
        Msg: Cardinal;
        CharCode: Word;
        Unused: Word;
        KeyData: Longint;
        Result: Longint;
    end;

```

А при манипуляциях пользователя мышью структура сообщения иная:

```

type
    TWMMouse = packed record
        Msg: Cardinal;
        Keys: Longint;
        case Integer of
            0: (
                XPos: Smallint;
                YPos: Smallint);
            1: (
                Pos: TSmallPoint;
                Result: Longint);
    end;

```

Дополнительное поле Result вставлено потому, что в некоторых случаях ядро Windows требует ответа на посланное сообщение. Например, на сообщение о перемещении мыши в режиме перетаскивания (Drag&Drop) программа должна отвечать, готова ли она принять «посылку».

В подавляющем большинстве случаев для обработки *сообщений* программист использует обработчики *событий* компонентов. Реализуется это следующим образом. Оконная процедура анализирует тип сообщения и определяет, создан ли в программе нужный обработчик события. Если создан, преобразует сообщение в наиболее удобную для обработки форму и передает управление обработчику, если нет (и после завершения работы обработчика), происходит стандартная (заданная по умолчанию) обработка. Еще раз подчеркну, что обработка сообщения реализуется всегда, независимо от того, создан ли для этих целей обработчик соответствующего события.

В некоторых специальных случаях программисту может понадобиться нестандартная обработка того или иного события. Для этого он должен создать метод объекта, объявив его с директивой **message**:

```
procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
```

В такого рода обработчиках обычно вначале реализуются нестандартные действия, а потом вызывается унаследованный обработчик, который совершает стандартную обработку:

```
procedure TForm1.WMPaint(var Msg: TWMPaint);
begin
    // Реализуем нестандартные действия
    ...
    inherited // Вызываем унаследованный обработчик
end;
```

Однако в некоторых случаях удобнее сначала реализовать стандартную обработку или даже вовсе отказаться от нее. Пример нестандартного обработчика сообщения Windows вы найдете в главе 21 (проект Chap_21\MovedWindow\MovWin.dpr).

В ряде случаев можно организовать нестандартную обработку в обработчике OnMessage объекта Application. Его заголовок таков:

```
procedure TForm1.AppMessageHandler(var Msg: TMsg;
                                     var Handled: Boolean);
```

Этот обработчик получает любое сообщение до какой-либо предварительной его обработки и в параметре Handled сообщает оконной процедуре, нуждается ли переданное ему сообщение в дополнительной обработке (False — нуждается). Название процедуры обработчика помещается в свойство OnMessage обычно в файле проекта перед обращением к методу Run:

```
program Project1;
...
begin
    ...
    Application.OnMessage := Form1.AppMessageHandler;
    Application.Run;
end.
```

ВНИМАНИЕ Программа обычно получает множество сообщений Windows, поэтому обработчик OnMessage является самой вызываемой процедурой вашей программы. Вряд ли стоит выполнять в нем хоть какую-то работу, требующую интенсивных вычислений, иначе ваша программа перестанет немедленно реагировать на сообщения и «зависнет».

Посылка сообщений

Программа не только получает, но также может посылать сообщения — самой себе или другой программе. Для этого в распоряжении программиста имеются методы Perform компонентов, методы Broadcast потомков класса TWinControl и, наконец, API-функции SendMessage и PostMessage.

Метод Perform есть у любого наследника класса TControl, в котором он описан следующим образом:

```
function TControl.Perform(Msg: Cardinal;
    WParam, LParam: LongInt): LongInt;
```

Обращение к этому методу удобно использовать для ускоренной обработки сообщения, так как в этом случае его доставкой Windows не занимается. Метод передает компоненту сообщение и ожидает результата его обработки.

Для передачи одного и того же сообщения сразу группе компонентов все наследники класса `TWinControl` (а именно этот класс является родителем всех компонентов-контейнеров) имеют метод `Broadcast`:

```
procedure TwinControl.Broadcast(var Message);
```

Этот метод также ожидает результата обработки сообщения каждым помещенным в контейнер компонентом. Если какой-то компонент в поле `Result` переданного ему сообщения возвращает не ноль, рассылка сообщения остальным компонентам прекращается, и метод завершает свою работу.

Для посылки сообщения с использованием механизма Windows предназначены следующие функции:

```
function SendMessage(hWnd: HWND; Msg: UINT; WParam: WPARAM;
    LParam: LPARAM): LRESULT;
```

```
function PostMessage(hWnd: HWND; Msg: UINT; WParam: WPARAM;
    LParam: LPARAM): BOOL;
```

Функции имеют одинаковый набор параметров и отличаются тем, что первая ожидает окончания обработки переданного сообщения и возвращает его результат, а вторая сразу же возвращает управление вызывающей программе и лишь сообщает о том, смогла ли она поместить сообщение в очередь. Замечу, что параметр `WPARAM` в первом случае обычно имеет значение 1, а во втором 0: так Windows извещает программу, ждет операционная система немедленного ответа или нет.

Пользовательские сообщения

Программа может создать и передать другой программе (или другому окну своей программы) нестандартное для Windows (пользовательское) сообщение. С этой целью Windows резервирует диапазон значений кодов сообщений `WM_USER...$7FFFF`. Например:

```
const
    SX_MYMESSAGE = WM_USER+1;
begin
    ...
    SendMessage(SomeForm.Handle, SX_MYMESSAGE, 1, 0);
    ...
end;
```

Чтобы оконная процедура формы-адресата смогла правильно отреагировать на такое сообщение, в ней следует предусмотреть соответствующий обработчик:

```
TSomeForm = class (TForm)
...
private
```



```

procedure SXMyMessage(var Msg: TMessage); message
    SX_MYMESSAGE;
...
end;

procedure SomeRorm.SXMyMessage(var Msg: TMessage);
begin
    // Код реакции на пользовательское сообщение
end;

```

ВНИМАНИЕ Никогда не посылайте сообщение, если вы не уверены, что адресат сможет его обработать.

Функции API

Функции API (Application Program Interface — интерфейс прикладных программ) представляют собой «ключи» от всех возможностей Windows. Можно с уверенностью утверждать, что знание этих функций и особенностей работы с ними прямо пропорционально опыту и способностям программиста, работающего в среде Windows.

Эти функции сосредоточены в многочисленных библиотеках, расположенных в папках System32, System и других, указанных в переменной окружения PATH. Многие из них описаны в модуле WINDOWS и, следовательно, готовы к немедленному использованию в программах Delphi, так как ссылка на этот модуль в обязательном порядке помещается в предложение **uses** любого модуля.

Например, в любом нужном месте программы вы можете потребовать перезагрузки Windows с помощью следующего кода:

```
Win32Check(ExitWindowsEx(EWX_REBOOT, 0));
```

Этот код иллюстрирует предпочтительный способ вызова функций API, в которых могут возникнуть исключительные ситуации (функция ExitWindowsEx в Windows NT/2000/XP может вызывать исключение, если пользователь не обладает специальным правом SE_SHUTDOWN_NAME). Функция Win32Check проверяет код завершения вызова ExitWindowsEx и обращается к функции GetLastError, если возникла ошибка. Последняя дает развернутое сообщение о характере ошибки.

Однако львиная доля функций API не описана в модуле WINDOWS и требует описания в том модуле, в котором реализуется обращение к ним. Например, функции timeSetEvent и timeKillEvent используют внутренний системный таймер Windows, с помощью которого реализуется механизм вытесняющей многопоточности, то есть принудительного прерывания работы потока команд по истечении выделенного ему кванта времени. Перед использованием их следует описать:

```

function timeSetEvent(uDelay, uReolution: UINT;
    lpTimeProc: Pointer; dwUser: DWORD; fuEvent: UINT):
    Integer;

```

```
stdcall; external 'winmm.dll';
```

```
function timeKillEvent(uID: UINT): Integer;  
stdcall; external 'winmm.dll';
```

ВНИМАНИЕ Все функции API следуют соглашению **stdcall** об использовании регистров и стека. Кроме того, при работе под управлением Windows NT/2000/XP нужно указывать имя библиотеки с расширением DLL (при работе под Windows 95/98 расширение можно опускать).

Часть IV

Расширенные возможности Delphi

В трех первых частях книги достаточно полно описываются базовые возможности Delphi, без знания которых невозможно создание даже относительно несложных программ среднего объема. В этой части вы познакомитесь с некоторыми дополнительными возможностями Delphi, освоение которых является обязательным условием успешного программирования в этой среде.

В главе 23 рассматривается методика создания и использования динамически подключаемых библиотек (DLL). Эти библиотеки широко используются в Windows как средство обмена готовыми программными решениями, написанными, возможно, на других языках программирования. Фактически, сама операционная система Windows состоит из нескольких десятков библиотек DLL, написанных, преимущественно, на языке C++.

В главах 24 и 25 вы познакомитесь с технологией COM, которая лежит в основе многих возможностей Windows. Собственно говоря, интерфейсы, рассматриваемые в главе 24, не являются частью этой технологии, но широко в ней используются. Изучив этот материал, вы сможете подключать к вашей программе многочисленные COM-серверы, такие как текстовый процессор Word, табличный процессор Excel и т. п.

Как уже отмечалось, особенностью версии Delphi 7 Studio является большое количество дополнительных утилит, которые поставляются вместе с ней, но не являются ее неотъемлемой частью. Изучение всех этих утилит выходит за рамки книги. Тем не менее я не мог не познакомить вас с одной из них — ModelMaker. В главе 26 вы изучите некоторые возможности этой системы автоматизированного программирования. Попутно в этой главе рассматриваются основные проблемы создания собственных компонентов.

Глава 23

Создание и использование DLL

Динамически подключаемые библиотеки (Dynamic Link Library, DLL) предоставляют универсальный механизм интегрирования в вашу программу процедур и функций, написанных другими программистами и, в общем случае, на других, нежели Delphi, языках программирования.

DLL реализуются в виде исполняемых модулей, содержащих готовые к работе процедуры, функции и/или ресурсы. С точки зрения программиста, есть много общего между DLL и обычными для Delphi модулями, так как в конечном счете и библиотеки, и модули поставляют подпрограммы, избавляющие программиста от написания собственного кода. Но есть и принципиальные отличия. Главным из них является то, что DLL не в состоянии поставлять в программу переменные, константы и типы, ведь создатели DLL могут использовать нетипизированные языки программирования, например, язык ассемблера. В результате библиотеки DLL не могут экспортировать в программу столь необходимые сегодня программисту классы — для этого требуются пакеты.

Другим важным отличием является способ связывания экспортируемых подпрограмм с основной программой. Модули связываются с программой на этапе компоновки, то есть *статически*. Если загружены две программы, использующие одни и те же модули, в памяти будут два экземпляра одинаковых фрагментов кода. В отличие от этого DLL подключаются к программе в момент ее исполнения, то есть *динамически*. Если, опять же, две программы используют одну и ту же библиотеку DLL, в памяти будет лишь один экземпляр разделяемого программой кода. Следует уточнить, что речь идет о физической памяти компьютера. Поскольку каждая программа получает в свое распоряжение виртуальное адресное пространство (подробнее о процессах и памяти рассказывается в главе 22), в эти пространства будут отображаться столько *образов* библиотеки DLL, сколько программ ее используют (см. раздел «Отображение файлов в память» в главе 10).

Динамическое подключение дает DLL еще одно немаловажное преимущество над модулями: изменение любой библиотеки DLL в большинстве случаев не требует перекомпиляции использующей ее программы.

Реализация

Для создания DLL в Delphi введено зарезервированное слово **Library** (библиотека), которым должен начинаться текст библиотеки. За словом **Library** следует правильный идентификатор, но в отличие от объявления модуля он не обязан совпадать с именем файла: имя DLL определяется именем DLL-файла, а не идентификатором, следующим за словом **Library**.

Структура текста DLL повторяет структуру обычной программы с тем исключением, что раздел исполняемых операторов в DLL играет ту же роль, что и инициализирующая часть модуля: операторы этой части исполняются только один раз в момент загрузки библиотеки в память. Каждое очередное обращение с требованием загрузить библиотеку наращивает на единицу ее счетчик ссылок, но не приводит к выполнению операторов исполняемой части.

В разделе описаний DLL могут объявляться типы (в том числе и классы), константы и переменные, но они остаются скрытыми от вызывающей программы и могут использоваться только внутри DLL. В разделе описаний помимо стандартных для обычной программы объявлений используется специальный раздел объявления экспортируемых подпрограмм. Этот раздел начинается зарезервированным словом **Exports**, за которым через запятую перечисляются имена экспортируемых подпрограмм, например:

```
Library MyLibrary;
function MyFunc (...):...;
begin
    ...
end;
procedure MyProc;
begin
    ...
end;
Exports
    MyFunc, MyProc;
begin
end.
```

Раздел **Exports** помогает компилятору и компоновщику создать специальный заголовок DLL-модуля, в котором перечисляются имена подпрограмм и адреса их точек входа. В DLL может быть несколько списков **Exports**, но перечисляемые в них подпрограммы должны быть описаны где-то выше по тексту библиотеки.

Помимо имени подпрограммы в заголовок DLL помещается также ее порядковый номер, точнее, присвоенный ей целочисленный индекс. Это позволяет вызывающей программе ссылаться не на имя, а на индекс подпрограммы и тем самым уменьшить затраты времени на установление с ней связи. Индекс присваивается подпрограмме в соответствии с порядком ее появления в списках **Exports**: первая подпрограмма в первом списке получает индекс 0, следующая — 1 и т. д. Программист может изменить такой порядок индексации и явно указать индекс подпрограммы, добавив за ее именем в списке **Exports** слово **index** и целое число без знака в диапазоне от 0 до 32 767:

Exports

```
MyFunc index 1, MyProc index 2;
```

ВНИМАНИЕ

Корпорация Microsoft объявила о том, что в дальнейшем она, возможно, откажется от поддержки индексов в DLL и рекомендовала вызывать подпрограммы только по именам.

Программист может определить такое внешнее имя экспортируемой подпрограммы, которое будет отличаться от ее настоящего имени. Для этого в списке **Exports** добавляется слово `name` и внешнее имя в апострофах:

Exports

```
MyFunc index 1 name 'NEWFUNC';
```

ПРИМЕЧАНИЕ

Внешние имена критичны к регистру букв, так что NEWFUNC и NewFunc — это разные имена.

Вызывающая программа может ссылаться либо на имя экспортируемой подпрограммы, либо на ее индекс. При вызове по имени программа просматривает имена в таблице имен в поисках нужного. Так как имена могут состоять из длинных наборов символов, а самих имен в таблице может быть много, процесс поиска имени медленнее, чем процесс поиска индекса.

Все операторы, расположенные после слова **begin** и до завершающего слова **end**, играют ту же роль, что и операторы инициализирующей части модуля: они выполняются один раз в момент загрузки DLL. В отличие от модуля, в DLL нет завершающей секции.

Замечу, что Delphi не компилирует библиотеку DLL автоматически в режимах MAKE или BUILD, так как справедливо рассматривает ее как другую программу, никак не связанную в момент компиляции с основной программой.

Пример создания DLL

Рассмотрим пример создания DLL, в котором иллюстрируются различные приемы объявления экспортируемых подпрограмм. Для примера выбран модуль `Cmplx`, описанный в разделе «Доступ к объявленным в модуле объектам» главы 11. В его состав входят 4 процедуры, реализующие действия с комплексными числами. Вариант соответствующей библиотеки DLL показан в листинге 23.1 (см. проект `Chap_23\DLL Cmplx\Cmplx.dpr`).



Для создания заготовки библиотечного модуля выберите команду **File ▶ New ▶ Other** и в окне хранилища щелкните на значке DLL. В ответ Delphi откроет специальное окно проекта с длинным комментарием, в котором указывается на необходимость вставить ссылку на модуль `ShareMem`, если библиотека экспортирует длинные строки в параметрах обращения к подпрограммам или как результат функций. Эта ссылка должна быть первой как в предложении **Uses** библиотеки, так и в предложении **Uses** файла проекта программы, которая использует эту библиотеку. Если подпрограммы библиотеки экспортируют строки `ShortString` или `PChar`, ссылаться на `ShareMem` не обязательно. Сразу же сохраните

проект под именем Cmplx, чтобы среда Delphi автоматически исправила имя библиотеки в предложении **Library**.

Листинг 23.1. Библиотека DLL, реализующая действия с комплексными числами¹

```

Library Cmplx;

uses
    SysUtils,
    Classes;

{$R *.RES}

type
    TComplex = record
        Re, Im: Real;
    end;

function AddC(x, y: TComplex): TComplex; stdcall;
begin
    Result.Im := x.Im + y.Im;
    Result.Re := x.Re + y.Re
end;

function SubC(x, y: TComplex): TComplex; stdcall;
begin
    Result.Im := x.Im - y.Im;
    Result.Re := x.Re - y.Re
end;

function MulC(x, y: TComplex): TComplex; stdcall;
begin
    Result.Re := x.Re * y.Re + x.Im * y.Im;
    Result.Im := x.Re * y.Im - x.Im * y.Re
end;

function DivC(x, y: TComplex): TComplex; stdcall;
var
    z: Real;
begin
    z := sqr(y.Re) + sqr(y.Im);
    Result.Re := (x.Re * y.Re + x.Im * y.Im) / z;
    Result.Im := (x.Re * y.Im - x.Im * y.Re) / z
end;

```

¹ Все представленные в книге тексты программ можно загрузить с сайта издательства по адресу <http://www.piter.com/download> — Примеч. ред.

Exports

```
AddC index 1 name 'ADDC' resident,
SubC index 2,
MulC index 3,
DivC index 4;
```

begin**end.**

Обратите внимание: все функции нашей библиотеки DLL используют директиву **stdcall**, которая обеспечивает совместимость новых функций с функциями API 32-разрядных версий Windows. Мы могли бы не указывать эту директиву, тогда компилятор использовал бы более эффективное соглашение, реализуемое директивой **register**, но обращение к нашей библиотеке DLL из программ, написанных на других языках программирования, в общем случае стало бы невозможным.

СОВЕТ Если вы создаете DLL для «внешнего» использования (вне Delphi), объявляйте подпрограммы с директивой **stdcall** или **safecall**.

Для использования подпрограмм из DLL необходимо описать их как внешние, добавив за словом **External** имя библиотеки в апострофах:

```
Procedure MyProc; External 'MyDLL';
```

ВНИМАНИЕ Если ваш компьютер работает под управлением Windows 95/98, расширение DLL можно опускать (как в только что приведенном примере), однако при работе с Windows NT/2000/XP нужно указывать полное имя библиотеки с расширением DLL.

Как уже отмечалось (см. выше подраздел «Реализация»), подпрограмма вызывается по имени или по индексу. В нашем примере из библиотеки MyDLL вызывается подпрограмма с внешним именем 'MyProc'. Если нужно сослаться на индекс подпрограммы, за именем библиотеки указывается слово **index** и сам индекс:

```
Procedure MyProc; External 'MyDLL.dll' index 2;
```

В этом случае имя, под которым подпрограмма будет известна программе, может не совпадать с ее внешним именем, определенным в DLL. Впрочем, программист может и явно переопределить имя подпрограммы, даже если он ссылается на ее внешнее имя:

```
Procedure MyProc; External 'MyDLL.dll' Name 'ExtName';
```

В этом варианте предполагается, что экспортируется процедура с внешним именем 'ExtName'.

После любого из указанных выше объявлений экспортируемая подпрограмма становится доступной программе и может вызываться в ней как обычная подпрограмма Delphi.

Статическая загрузка

В следующей программе, представленной в листинге 23.2 (проект Chap_23\DLL Cmplx\TestDLLCmplx.dpr), используется библиотека Cmplx, описанная в листинге 23.1.

Листинг 23.2. Статическая загрузка DLL

```

type
  TComplex = record
    Re, Im: Real;
  end;
function ADDC(x,y:TComplex): TComplex;
  stdcall; External 'Cmplx.dll';
function SubC(x,y:TComplex): TComplex;
  stdcall; External 'Cmplx.dll';
function MulC(x,y:TComplex): TComplex;
  stdcall; External 'Cmplx.dll';
function DivC(x,y:TComplex): TComplex;
  stdcall; External 'Cmplx.dll';
procedure TfmExample.bbRunClick(Sender: TObject);
var
  x,y,z: TComplex;
procedure Output(Operation: Char);
  // Осуществляет нужное действие и выводит результат
  // в mmOutput
var
  S: String;
begin
  case Operation of
    '+': z := AddC(x,y);
    '-': z := SubC(x,y);
    '*': z := MulC(x,y);
    '/': z := DivC(x,y);
  end;
  S := '('+FormatFloat('+0.0000;-0.0000',x.re)+
    FormatFloat('+0.0000j;-0.0000j',x.im)+')'+
    Operation+
    '('+FormatFloat('+0.0000;-0.0000',y.re)+
    FormatFloat('+0.0000j;-0.0000j',y.im)+'='+'
    FormatFloat('+0.0000;-0.0000',z.re)+
    FormatFloat('+0.0000j;-0.0000j',x.im);
  mmOutput.Lines.Add(S);
end; // Output

begin // bbRunClick
  x.re := Random;
  x.im := Random;
  y.re := Random;
  y.im := Random;
  Output('+');
  Output('-');

```

```

Output('*');
Output('/');
mmOutput.Lines.Add('');
end;

```

```
end.
```

Обратите внимание: библиотечная функция `CmplxAdd` имеет внешнее имя `ADDC`. Именно так (прописными буквами) описана эта функция в приведенном выше примере. Если бы мы использовали показанный ниже вариант описания, компилятор не смог бы ее идентифицировать:

```

function AddC(x,y:TComplex): TComplex;
  stdcall; External 'Cmplx.dll';

```

ПРИМЕЧАНИЕ В некоторых случаях внутренние имена подпрограмм DLL могут содержать символы, недопустимые в правильных идентификаторах Delphi. Например, в идентификаторах C++ разрешается использовать символ `@`. Объявить такую подпрограмму, сославшись на ее имя, нельзя. В этом случае нужно использовать правильное с позиций Delphi имя, указав истинное имя подпрограммы за словом `name`. Например:

```

function ClientRevision: Cardinal;
external 'WTClient.dll' name '_WTclientRevision@0';

```

Динамическая загрузка

Описанный выше способ определения функций и процедур DLL (с помощью директивы **External**) заставит компилятор поместить в заголовок программы список всех библиотек DLL, и загрузчик загрузит их в память одновременно с загрузкой самой программы. Программа может загружать DLL и без директивы **External** с помощью трех стандартных функций: `LoadLibrary`, `GetProcAddress` и `FreeLibrary`. Листинг 23.3 (проект `Chap_23\DLL Cmplx\TestDLLCmplx2.dpr`) иллюстрирует технику такой загрузки DLL `Cmplx`.

Листинг 23.3. Динамическая загрузка DLL

```

type
  TComplex = record
    Re, Im: Real;
  end;

  TComplexFunc = function (x, y: TComplex): TComplex;
  stdcall;

procedure TfmExample.bbRunClick(Sender: TObject);
var
  x,y,z: TComplex;
  AddC,SubC,MulC,DivC: TComplexFunc;
  Handle: LongWord;

```

```

procedure Output (Operation: Char);
...// Тело процедуры Output остается без изменений
end; // Output

begin // bbRunClick
    // Загружаем библиотеку CMPLX.DLL
    Handle := LoadLibrary('Cmplx.dll');
    if Handle = 0 then
        begin
            ShowMessage('Не найдена библиотека CMPLX.DLL');
            Halt
        end;
    { Определяем адреса функций. Три первые вызываем по
    индексу, четвертую - по имени. При вызове по индексу
    младшее слово PChar должно содержать индекс, поэтому
    выполняем приведение типов: }
    @AddC := GetProcAddress(Handle, PChar(LongInt(1)));
    @SubC := GetProcAddress(Handle, PChar(LongInt(2)));
    @MulC := GetProcAddress(Handle, PChar(LongInt(3)));
    @DivC := GetProcAddress(Handle, 'DivC');

    x.re := Random;
    x.im := Random;
    y.re := Random;
    y.im := Random;
    Output('+');
    Output('-');
    Output('*');
    Output('/');
    mmOutput.Lines.Add('');
    // Освобождаем библиотеку
    FreeLibrary(Handle)
end;

```

Интерфейсный модуль

При вызове подпрограмм DLL в большинстве случаев бывает необходимо передавать структурированные параметры типа записей, как тип TComplex в предыдущем примере. Поскольку DLL не могут экспортировать типы, приходится объявлять эти типы в вызывающей программе. Если вы часто обращаетесь в своих программах к той или иной библиотеке DLL, удобно создать интерфейсный модуль, содержащий объявления как подпрограмм, так и связанных с ними типов. Пример представлен в листинге 23.4 (файл Chap_23\DLL Cmplx\Complx.pas):

Листинг 23.4. Интерфейсный модуль

```

Unit Complx;
Interface

```

```

type
  TComplex = record
    Re, Im: Real;
  end;
function AddC(x,y:TComplex): TComplex;
  stdcall; External 'Cmplx.dll';
function SubC(x,y:TComplex): TComplex;
  stdcall; External 'Cmplx.dll';
function MulC(x,y:TComplex): TComplex;
  stdcall; External 'Cmplx.dll';
function DivC(x,y:TComplex): TComplex;
  stdcall; External 'Cmplx.dll';

```

Implementation

end.

Такой интерфейсный модуль существенно упрощает разработку основной программы: в нашем примере (проект Chap_23\DLL Cmplx\TestDLLCmplx3.dpr) он обеспечивает такой же интерфейс к библиотеке Cmplx, как описанный выше модуль Cmplx к своим объектам.

Включение в библиотеку форм

Несмотря на то, что библиотека DLL не имеет собственной формы, она позволяет вызывать формы из связанных с библиотекой модулей. Для этого в библиотеке используется ссылка **uses** на связанные модули-формы и объявляются экспортируемые из DLL подпрограммы, в которых реализуется вызов соответствующих форм.

В листингах 23.5, 23.6 и 23.7 (см. библиотеку Chap_23\Forma into DLL\DLLWithForm.dpr и тестовый пример Chap_23\Forma into DLL\TestDLLForm.dpr) иллюстрируется техника включения в DLL формы и использования этой формы в вызывающей программе.

Листинг 23.5. Текст DLL

```

library DLLWithForm;

uses
  SysUtils,
  Classes,
  DLLFormU in 'DLLFormU.pas' {DLLForm};

{$R *.RES}

exports
  ShowModalForm, ShowForm, FreeForm;

```

```
begin
end.
```

Листинг 23.6. Текст формы в DLL

```
unit DLLFormU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons;

type
  TDLLForm = class(TForm)
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    procedure FormClose(Sender: TObject;
      var Action: TCloseAction);
  private
    { Private declarations }
    CallForm: THandle; // Дескриптор вызывающей формы
  public
    { Public declarations }
  end;

  // Объявление экспортируемых подпрограмм
  function ShowModalForm: Integer;
  procedure ShowForm(aHandle: THandle);
  procedure FreeForm;

var
  DLLForm: TDLLForm;

implementation

{$R *.DFM}

function ShowModalForm: Integer;
// Модальный вызов
begin
  DllForm := TDllForm.Create(Application);
  Result := DLLForm.ShowModal;
  DLLForm.Free;
end;

procedure ShowForm(Appl, Form: THandle);
// Немодальный вызов
```

```

begin
  Application.Handle := Appl;  // Замена объекта
                               // Application
  DllForm := TDllForm.Create(Application);
  // Запоминаем дескриптор вызывающего окна для отправки
  // ему сообщения о закрытии
  CallForm := Form;
  DLLForm.Show
end;

procedure FreeForm;
// Удаление формы
begin
  DLLForm.Free
end;

procedure TDLLForm.FormClose(Sender: TObject;
var Action: TCloseAction);
begin
  if CallForm<>0 then
    SendMessage(CallForm, wm_User, 0, 0)
end;

end.

```

Листинг 23.7. Текст вызывающей программы

```

unit TestMainU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TTestMain = class (TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }

```

```

    procedure WMUser(var Msg: TMessage); message WM_USER;
end;

var
    TestMain: TTestMain;

implementation

{$R *.DFM}

function ShowModalForm: Integer; External
    'DLLWithForm.dll';
procedure ShowForm(Appl, Form: THandle);
    External 'DLLWithForm.dll';
procedure FreeForm; External 'DLLWithForm.dll';

procedure TTestMain.Button1Click(Sender: TObject);
// Модальный вызов
begin
    Button2.Enabled := False;
    label1.Caption := 'ModalResult = ' +
        IntToStr(ShowModalForm);
    label1.Show;    // Показываем результат вызова
    Button2.Enabled := True
end;

procedure TTestMain.Button2Click(Sender: TObject);
// Немодальный вызов
begin
    Button1.Enabled := False;
    Button2.Enabled := False;
    Button3.Enabled := True;
    label1.Hide;
    ShowForm(Application.Handle, Self.Handle);
end;

procedure TTestMain.Button3Click(Sender: TObject);
// Закрывать форму
begin
    FreeForm;
    Button1.Enabled := True;
    Button2.Enabled := True;
    Button3.Enabled := False;
end;

procedure TTestMain.WMUser(var Msg: TMessage);
// Сообщение из формы к DLL о закрытии формы

```

```
begin
  Button3.Click
end;

end.
```

Модуль формы `DLLForm`, помещенной в DLL, ссылается на стандартный модуль `Forms` и, таким образом, получает свой глобальный объект `Application`, который ничего «не знает» о глобальном объекте вызывающей программы (см. раздел «Глобальные объекты» в главе 22). В режиме модального вызова это не имеет особого значения, так как модальное окно блокирует работу вызывающей программы. В режиме немодального вызова следует синхронизировать действия объектов, в противном случае свертывание главного окна, например, не приведет к свертыванию окна DLL. Синхронизация достигается тем, что дескриптор объекта `Application` DLL заменяется соответствующим дескриптором вызывающей программы.

При показе формы в немодальном режиме она может быть закрыта щелчком на собственной системной кнопке закрытия. В этом случае она должна каким-то образом известить вызывающую программу об этом событии. Для этого используется стандартный механизм послыки вызывающей форме сообщения `Windows`. Подробнее о сообщениях `Windows` рассказывается в главе 22, здесь я лишь отмечу, что сообщение должно иметь адрес, в роли которого используется дескриптор окна, получающего сообщение. Вот почему дескриптор вызывающего окна передается вторым параметром вызова функции `ShowForm` в DLL и запоминается в поле `CallForm`. Обработчик события `OnClose` формы проверяет это поле и, если оно определено, посылает вызывающему окну сообщение с индексом `WM_USER`. В вызывающей программе предусмотрен обработчик этого сообщения, в котором реализуются необходимые действия.

Особенности использования модулей в DLL

В DLL можно ссылаться на любые модули Delphi в предложении **uses**. Однако следует помнить, что инициализирующие и завершающие секции этих модулей не будут исполняться в момент загрузки программы в память и при завершении ее работы, если на те же модули не ссылается основная программа или один из ее модулей. В некоторых случаях это может привести к обращению внутри DLL к объектам, не созданным в инициализирующей части модуля, и, как следствие, к краху программы.

Если, например, библиотека работает с технологией автоматизации OLE (см. главу 25), она должна ссылаться на модуль `ComObj`, в инициализирующей части которого создается объект компонентного класса для доступа к дуальным интерфейсам (см. главу 24). При разработке DLL для использования в среде Delphi эта проблема легко решается добавлением ссылки на модуль (модули) в главной программе или любом ее модуле. Однако если DLL разрабатывается для продажи или работы с другими системами программирования (Visual Basic, C++, JBuilder и т. п.), эта проблема в большинстве случаев неразрешима, так как в инициализирующей части модуля часто используются обращения к полям и методам, закрытым для доступа вне модуля.

Глава 24

Интерфейсы

Интерфейсы играют главную роль в технологии СОМ (см. следующую главу) и связанных с ней технологиях удаленного доступа, то есть технологиях доступа к объектам, расположенным (и выполняющимися) на другой машине. Их основное назначение — описать свойства, методы и события удаленного объекта в терминах машины клиента, то есть на используемом при разработке клиентского приложения языке программирования. С помощью интерфейсов программа клиента обращается к удаленному объекту так, как если бы он был ее собственным объектом.

Создание и использование интерфейсов

Описание интерфейса

Описание интерфейсов представляет собой частный случай описания классов. Интерфейсы объявляются с помощью зарезервированного слова **interface**. Например:

```
type
  IEdit = interface
    procedure Copy; stdcall;
    procedure Cut; stdcall;
    procedure Paste; stdcall;
    function Undo: Boolean; stdcall;
end;
```

Такое объявление эквивалентно описанию абстрактного класса в том смысле, что не требует расшифровки объявленных в интерфейсе свойств и методов. В отличие от классов, интерфейс не может содержать поля, и, следовательно, объявляемые в нем свойства в разделах **read** и **write** могут ссылаться только на методы. Все объявляемые в интерфейсе члены размещаются в единственной секции **public**. Методы не могут быть абстрактными (**abstract**), виртуальными (**virtual**), динамическими (**dynamic**) или перекрываемыми (**override**).

ПРИМЕЧАНИЕ По принятому в Delphi соглашению имена интерфейсов начинаются буквой I.

Идентификатор интерфейса

Если интерфейс предполагается использовать в технологии COM, он обязательно должен снабжаться так называемым глобально уникальным идентификатором (Globally Unique Identifier, GUID). Этот идентификатор (Interface Identifier, IID) создается по специальной технологии, гарантирующей ничтожно малую вероятность совпадения вновь создаваемого IID с любым ранее созданным¹.

ПРИМЕЧАНИЕ В среде Delphi для получения GUID достаточно нажать комбинацию клавиш Ctrl+Shift+G. Для программного получения GUID следует обратиться к функции API CoCreateGUID.

Любой GUID сохраняется в структуре следующего вида:

```
type
  PGUID = ^TGUID;
  TGUID = record
    D1: LongWord;
    D2: Word;
    D3: Word;
    D4: array [0..7] of Byte;
  end;
```

Поскольку оперировать подобной структурой сложно, Delphi позволяет записывать ее в виде такой строки:

```
'{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}'
```

Здесь X — шестнадцатеричная цифра.

Идентификатор интерфейса записывается сразу за его начальным объявлением. Например:

```
type
  IEdit = interface
    ['{BCC5C45F-AC0E-4FC3-9F8F-6105448423EA}']
    procedure Copy; stdcall;
    procedure Cut; stdcall;
    procedure Paste; stdcall;
    function Undo: Boolean; stdcall;
```

Программист может объявлять типизированные константы типа TGUID, например:

¹ GUID представляет собой 128-разрядное целое число. Его уникальность обеспечивается достаточно большим числом и великолепным алгоритмом его генерации, в котором используются текущие дата и время, тактовая частота процессора и номер сетевой карты. Наличие в компьютере сетевой карты гарантирует уникальность GUID, так как по существующему отраслевому стандарту каждая сетевая карта имеет уникальный идентификатор.

```
const IID_IPaint: TGUID =
  ['{A4AFEB61-7705-11D2-8B41-444553540000}'];
```

Константы GUID могут использоваться вместо имен интерфейсов при вызове подпрограмм. Например, два следующих обращения идентичны:

```
procedure Paint(const IID: TGUID);
...
Paint(IPaint);
Paint(IID_Paint);
```

Компонентный класс

Интерфейсы не могут иметь конструкторов и деструкторов, так как описываемые в них методы реализуются только в рамках поддерживающих их классов, которые называются *компонентными* (*co-class*).

Если компонентный класс поддерживает интерфейс, имя этого интерфейса указывается при объявлении класса в списке его родителей:

```
TEditor = class(TInterfacedObject, IEdit)
procedure Copy; stdcall;
procedure Cut; stdcall;
procedure Paste; stdcall;
function Undo: Boolean; stdcall;
end;
```

В отличие от обычного класса, компонентный класс может иметь более одного родительского интерфейса:

```
type
  IMyInterface = interface
    procedure Delete; stdcall;
  end;

  TMyEditor = class(TInterfacedObject, IEdit, IMyInterface)
    procedure Copy; stdcall;
    procedure Cut; stdcall;
    procedure Paste; stdcall;
    function Undo: Boolean; stdcall;
    procedure Delete; stdcall;
  end;
```

В любом случае в разделе реализации компонентного класса необходимо описать соответствующие интерфейсные методы. Пусть, например, объявлен следующий интерфейс:

```
IPaint = interface
  procedure CirclePaint(Canva: TCanvas; X,Y,R: Integer);
  procedure RectPaint(Canva: TCanvas; X1,Y1,X2,Y2:
    Integer);
end;
```

Пусть также объявлен поддерживающий его компонентный класс:

```
TPainter = class (TInterfacedObject, IPaint)
  procedure CirclePaint (Canva: TCanvas; X,Y,R: Integer);
  procedure RectPaint (Canva: TCanvas; X1,Y1,X2,Y2:
    Integer);
end;
```

Тогда в разделе **implementation** следует указать реализацию методов:

```
procedure TPainter.CirclePaint (Canva: TCanvas; X,Y,R:
  Integer);
begin
  with Canva do
    Ellipse(X, Y, X+2*R, Y+2*R);
end;
```

```
procedure TPainter.RectPaint (Canva: TCanvas;
  X1,Y1,X2,Y2: Integer);
begin
  with Canva do
    Rectangle(X1, Y1, X2, Y2)
end;
```

Теперь можно объявить интерфейсный объект класса IPaint, чтобы с его помощью нарисовать окружность и квадрат:

```
procedure TForm1.PaintBox1Paint (Sender: TObject);
var
  Painter: IPaint;
begin
  Painter := TPainter.Create;
  Painter.CirclePaint (PaintBox1.Canvas,10,0,10);
  Painter.RectPaint (PaintBox1.Canvas,40,0,60,20);
end;
```

Несмотря на то, что интерфейс всегда объявляется до объявления поддерживающего его компонентного класса и, следовательно, известен компилятору, его методы обязательно должны быть перечислены в объявлении класса. В нашем случае простое указание классов, как это показано ниже, было бы ошибкой: компилятор потребовал бы вставить описание методов CirclePaint и RectPaint.

```
type
  TPainter = class (TInterfacedObject, IPaint)
end;
```

Класс TInterfacedObject и интерфейс IUnknown

Подобно тому как все классы в Delphi порождены от единственного родителя TObject, все компонентные классы порождены от общего предка TInterfacedObject. Этот предок умеет распределять память для компонентных объектов и использует глобальный интерфейс IUnknown:

```
type
  TInterfacedObject = class (TObject, IUnknown)
```

```

private
    FRefCount: Integer;
protected
    function QueryInterface(const IID: TGUID;
        out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
public
    property RefCount: Integer read FRefCount;
end;

```

Интерфейс IUnknown, в свою очередь, является базовым для создания любого интерфейса. Он объявлен так:

```

type
    IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID out: Obj):
        Integer;
        stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;

```

В рамках этого интерфейса декларируются три метода. Метод `_AddRef` увеличивает на 1 счетчик ссылок на интерфейс, в то время как метод `_Release` уменьшает на 1 этот счетчик. Метод `_AddRef` пользователь вызывает всякий раз, когда он хочет получить доступ к интерфейсу, а метод `_Release` вызывается, когда необходимость в интерфейсе отпадает. Как только в счетчике оказывается ноль, соответствующий объект интерфейса (компонентного класса) перестает существовать и сам себя выгружает из памяти. Таким образом, интерфейсы (компонентные классы) порождают объекты с *управляемым временем жизни*. Метод `QueryInterface` предназначен для того, чтобы пользователь выяснил, поддерживает ли компонентный объект нужный интерфейс IID. Если это так, становится действительной ссылка `Obj` на нужный компонентный объект, и метод возвращает ноль.

ПРИМЕЧАНИЕ

Опытные программисты, использующие другие языки программирования, знают, что название метода не может начинаться символом подчеркивания. Сделано это для того, чтобы скрыть от программиста этот метод. Таким образом, ни один из методов интерфейса IUnknown не должен вызываться явно (хотя Delphi и предоставляет такую возможность).

Псевдонимы методов

Методы компонентного класса могут исполняться под именами, отличными от имен методов интерфейса. Для этого при описании компонентного класса за именем интерфейсного метода ставится знак равенства и имя метода, который будет вызываться вместо него:

```

TPainter = class (TInterfacedObject, IPaint)
  procedure IPaint.CirclePaint = CPaint;
  procedure IPaint.RectPaint = RPaint;
  procedure CPaint (Canva: TCanvas; X,Y,R: Integer);
  procedure RPaint (Canva: TCanvas; X1,Y1,X2,Y2: Integer);
end;

```

Приведение интерфейсных типов

К интерфейсному объекту можно применить оператор приведения типов **as**, чтобы использовать нужный интерфейс:

```

procedure PaintObjects (P: TInterfacedObject)
var
  X: IPaint;
begin
  try
    X := P as IPaint;
    X.CirclePaint (PaintBox1.Canvas, 0, 0, 20)
  except
    ShowMessage ('Объект не поддерживает интерфейс IPaint')
  end
end;

```

Встретив такое присваивание, компилятор создаст код, с помощью которого вызывается метод `QueryInterface` интерфейса `IUnknown` с требованием вернуть ссылку на интерфейс `IPaint`. Если объект не поддерживает указанный интерфейс, возникает исключительная ситуация.

Уполномоченные свойства

С помощью зарезервированного слова **implements** программист может делегировать какому-либо свойству некоторого класса полномочия интерфейса, сделав это свойство уполномоченным. Это свойство должно иметь тип интерфейса или класса. Если свойство имеет тип интерфейса, имя этого интерфейса должно указываться в списке родителей класса, как если бы это был интерфейсный класс:

```

type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;

  TMyClass = class (TObject, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface
      implements IMyInterface;
  end;

```

Обратите внимание: в этом примере класс `TMyClass` *не является компонентным* классом, то есть классом, в котором исполняются методы P1 и P2. Однако если из него убрать определение уполномоченного свойства `MyInterface`, он станет компонентным и в нем должны быть описаны методы интерфейса `IMyInterface`.

Уполномоченное свойство обязательно должно иметь часть **read**. Если оно имеет тип класса, класс, в котором оно объявлено, не может иметь других уполномоченных свойств.

Объекты автоматизации и интерфейс IDispatch

В технологиях OLE/COM (см. главу 25) активно используются так называемые *объекты автоматизации* (*Automation objects*). Эти объекты представляют собой экземпляры компонентных классов, родительским интерфейсом которых является специальный интерфейс `IDispatch`. Отличительной особенностью интерфейса `IDispatch` является то обстоятельство, что методы объектов автоматизации никогда не вызываются напрямую, но всегда — с помощью метода `Invoke` интерфейса `IDispatch`.

Для объявления класса автоматизации используется специальное зарезервированное слово **dispinterface**, а перечисляемые в объявлении методы и свойства должны снабжаться целочисленными идентификаторами, которые вставляются в конце описания методов (свойств) после зарезервированных слов **dispid**:

```
type
  IStringsDisp = dispinterface
    ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
  property ControlDefault[Index: Integer]: OleVariant
    dispid 0; default;
  function Count: Integer; dispid 1;
  property Item[Index: Integer]: OleVariant dispid 2;
  procedure Remove(Index: Integer); dispid 3;
  procedure Clear; dispid 4;
  function Add(Item: OleVariant): Integer; dispid 5;
  function _NewEnum: IUnknown; dispid -4;
end;
```

С помощью зарезервированного слова **dispid** методу (свойству) интерфейса ставится в соответствие целочисленный номер. Это дает возможность языкам программирования, не поддерживающим указатели (`Visual Basic`, `Java` и др.), обращаться к методам и свойствам интерфейсных объектов по их именам. Для этого объект, имеющий интерфейс, родителем которого является **dispinterface**, автоматически создает таблицу имен его методов и свойств с указанием идентификаторов **dispid**. При обращении к методу по имени компилятор создает код вызова специального метода `IDispatch.GetIDsOfName`. Этот метод отыскивает в таблице имен нужное имя и возвращает соответствующий идентификатор **dispid** (если имя не найдено, возникает исключительная ситуация). Затем программа

использует полученный идентификатор как один из аргументов вызова метода `IDispatch.Invoke`. Этот метод отыскивает описание вызываемого метода в специальной таблице, проверяет количество и тип передаваемых методу параметров вызова и, если все правильно, вызывает его.

В отличие от обычного компонентного класса, класс автоматизации не может иметь родительского класса, и поэтому за словом **dispinterface** нельзя указать список родителей. Идентификаторы методов (свойств) должны быть уникальными в пределах объявления класса. Все возвращаемые функциями и свойствами результаты, а также все параметры обращения к методам должны иметь один из следующих типов: `Byte`, `Currency`, `Real`, `Double`, `LongInt`, `Integer`, `Single`, `SmallInt`, `AnsiString`, `WideString`, `TDateTime`, `Variant`, `OleVariant`, `WordBool` или любой интерфейсный тип. За исключением директивы **default**, которую можно указать для свойства-массива, никакие другие директивы в объявлении методов и свойств не допускаются.

Для доступа к объектам автоматизации используются переменные типа `variant`. Инициализация такой переменной осуществляется вызовом функции `CreateOleObject`, определенной в модуле `ComObj`. Эта функция возвращает ссылку на интерфейс `IDispatch`, с помощью которой можно обращаться к методам и свойствам класса автоматизации так, как если бы они были методами и свойствами варианта. Например:

```
Uses ComObj;
...
var
  Word: Variant;
begin
  Word := CreateOleObject('Word.Basic');
  Word.FileNew('Normal');
  Word.Insert('Первая строка'#13);
  Word.Insert('Вторая строка'#13);
  Word.FileSaveAs('c:\temp\test.txt', 3);
end;
```

Параметром обращения к методу `CreateOleObject` является имя сервера автоматизации, которое должно быть предварительно зарегистрировано в реестре 32-разрядных версий Windows. Характерно, что методы сервера не известны на этапе компиляции программы, поэтому компилятор никак не контролирует правильность их вызовов. Названия методов не подчиняются правилам построения идентификаторов Delphi, и в них могут использоваться символы национальных алфавитов.

Передаваемые методам параметры могут быть позиционными и именованными. Позиционные параметры являются обычными для подпрограмм Delphi параметрами-значениями. Именованные параметры записываются в следующем виде:

```
<имя_параметра> := <значение>
```

Например, при обращении к методу `FileSaveAs` (см. выше) были использованы два позиционных параметра. Это же обращение можно было бы записать с использованием именованных параметров следующим образом:

```
Word.FileSaveAs(Format := 3, Name := 'c:\temp\test.txt');
```


Именованные параметры могут перечисляться в произвольном порядке, но обязательно после позиционных параметров, если те тоже присутствуют в обращении.

Интерфейсы — наследники от `IDispatch` — называются дуальными, так как они дублируют свойства и методы основного интерфейса. Иными словами, Delphi может выполнять одну и ту же работу, либо обращаясь к экземплярам компонентного класса и используя указатели на нужные методы, либо по имени — с помощью дуального интерфейса. В отличие от этого, в Visual Basic доступен только последний способ. Методы дуальных интерфейсов (кроме унаследованных от интерфейсов `IUnknown` и `IDispatch`) должны компилироваться в режиме **safecall**.

Глава 25

Технология COM

Технология COM (Component Object Model — компонентная модель объектов) является краеугольной технологией Windows. Ее истоки можно найти еще в Windows 3.x в виде технологии DDE (Dynamic Data Exchange — динамический обмен данными). Эта технология использовала механизм сообщений Windows для передачи данных от одной программы (клиента) в другую (сервер). Позднее эта технология стала основой технологии OLE (Object Linking and Embedding — связывание и внедрение объектов), в которой программы обменивались не абстрактными данными, а *объектами* в виде рисунков, фрагментов текста и т. п. Затем появилась технология OLE2, в которой инициатор общения (клиент) мог автоматически «разбудить» спящий сервер (то есть заставить его работать, даже если перед этим сервер не был загружен в память компьютера). Многомиллионные инвестиции, сделанные корпорацией Microsoft в развитие упомянутых технологий, привели, в конце концов, к появлению мощной технологии COM, которая с необычайной легкостью интегрирует программы MS Office, дает возможность любой другой программе (в том числе и созданной в Delphi) обращаться к потенциальными COM-серверам с требованием выполнить ту или иную работу, наконец, заставить работать COM-сервер установленный на другой машине (возможно, в другой части света)!

Добавление в COM двух важных составных частей — *сервера транзакций* Microsoft (Microsoft Transaction Server, MTS) и *очереди сообщений* Microsoft (Microsoft Message Queue, MSMQ) трансформировало COM в технологию COM+, которая стала органической составной частью Windows 2000/XP. Новый стандарт позволяет всем пользователям новейших версий Windows пользоваться преимуществами таких средств, как управление транзакциями, поддержка безопасности, удаленное администрирование и т. д.

Я уверен, что как бы плохо не относился тот или иной профессионал к продуктам Microsoft (а ругать эту корпорацию и ее основателя и руководителя Б. Гейтса уже давно стало «признаком хорошего тона»), он не может «не снять шляпу» перед этой технологией. Говорить о программировании в Delphi и не раскрыть хотя бы некоторых тайн COM было бы с моей стороны просто непростительно.

Некоторые детали технологии

Технология COM представляет собой строго регламентированную спецификацию, определяющую требования к общающимся программам. При соблюдении этих требований гарантировано взаимодействие любых программ независимо от ис-

пользовавшегося при их создании языка программирования и места выполнения программ — в одном процессе, в разных процессах на одном компьютере, наконец, на разных компьютерах.

Общающиеся программы называются клиентом и сервером. Клиент является инициатором общения. Он обращается к одной из служб (сервисов) сервера с требованием получения некоторых данных и/или выполнения некоторой работы с данными, которые передаются серверу. Службы сервера реализуются в виде одного или нескольких входящих в его состав объектов COM. Каждая служба описывается своим интерфейсом; один объект может содержать произвольное количество служб и, таким образом, специфицироваться множеством интерфейсов. Любой сервер содержит как минимум один объект COM. Он (сервер) реализуется в виде исполняемого файла или библиотеки DLL. Характерной особенностью технологии является автоматическая активизация сервера при обращении к нему клиента. Если сервер закончил обслуживание всех клиентов, он также автоматически выгружается из памяти.

В этом разделе рассматривается техника взаимодействия с помощью интерфейсов, механизм автоматической активизации сервера, а также некоторые другие детали COM, необходимые для понимания принципов и терминологии этой технологии.

Взаимодействие с помощью интерфейсов

Интерфейс представляет собой абстрактный класс, описывающий методы и свойства объекта (см. главу 24). Получив тем или иным способом ссылку на интерфейс объекта, клиент может обратиться к этим методам и свойствам так, как если бы объект был его (клиента) составной частью. Каждый интерфейс в рамках COM снабжается глобально-уникальным идентификатором (GUID), что гарантирует его неизменность и, таким образом, обеспечивает нужное обслуживание клиента независимо от возможной модификации объекта.

Любой объект имеет интерфейс IUnknown, с помощью которого решаются две важнейшие проблемы: корректное освобождение выделенных объекту системных ресурсов при его уничтожении и предоставление клиенту ссылки на экземпляр (объект) класса, в котором реализован нужный клиенту интерфейс (компонентный класс).

Суть первой проблемы состоит в том, что, в общем случае, клиент и сервер выполняются в разных процессах и, кроме того, сервер может одновременно обслуживать нескольких клиентов. Каждый процесс обслуживается своим менеджером памяти, поэтому клиент не может корректно освободить связанные с объектом ресурсы после того, как его запрос был удовлетворен. Более того, если, например, текстовый процессор Word одновременно обслуживает двух клиентов, то его удаление одним из них приведет к тому, что второй будет иметь ссылку на несуществующий объект и его попытка обратиться к методу или свойству процессора Word приведет к возникновению исключительной ситуации. Для решения этой проблемы интерфейс IUnknown содержит методы AddRef и Release. Первый автоматически вызывается в момент получения запроса от клиента, при его выполнении наращивается на единицу внутренний счетчик ссылок. Второй вызывается в мо-

мент завершения обслуживания клиента и уменьшает этот счетчик на единицу. Если счетчик содержит ноль, сам серверный объект вызывает собственный деструктор, поэтому связанные с объектом системные ресурсы освобождаются вполне корректно и только тогда, когда в услугах объекта не нуждается ни один клиент.

Для предоставления клиенту ссылки на нужную службу сервера в интерфейсе IUnknown имеется метод QueryInterface. Он получает от клиента идентификатор IID (Interface Identifier — так называется GUID для интерфейса) требуемого интерфейса, просматривает список всех интерфейсов объекта и, если нужный интерфейс обнаружен, возвращает указатель на реализующий его объект. Более подробно алгоритм его работы после того, как интерфейс найден, описан ниже.

1. Проверяется, не был ли ранее создан экземпляр компонентного класса (то есть объект, реализующий описанные в интерфейсе методы и свойства), если нет — вызывается конструктор класса.
2. Вызывается метод AddRef этого экземпляра (объекта), чтобы увеличить его счетчик ссылок на единицу.
3. В нетипизированную переменную Obj, передаваемую клиентом в качестве параметра обращения, помещается указатель на экземпляр компонентного класса, а в результат — значение S_OK.

Получив указатель на интерфейсный объект и зная собственно интерфейс (как — об этом говорится ниже), клиент может обращаться к свойствам и методам объекта, как к своим собственным.

Замечу, что указатель недействителен в адресном пространстве клиента (за исключением случая, когда объект COM содержится в DLL). Для разрешения клиенту использования ссылки на адресное пространство сервера в программе сервера создается специальный объект *заглушка* (*stub*) — полномочный представитель клиента в адресном пространстве сервера, а в программе клиента — объект *заместитель* (*proxy*), который является полномочным представителем сервера в адресном пространстве клиента. Оба объекта связываются друг с другом и преобразуют указатели: когда сервер передает клиенту указатель, заглушка заменяет его указателем, действительным в адресном пространстве клиента. Точно так же поступает и заместитель, осуществляющий обратное преобразование.

После завершения работы с сервером клиент обращается к его интерфейсному методу Release. При программировании в Delphi последнее делать не нужно, так как компилятор автоматически создает код завершения программы, в котором проверяется действительность указателей на интерфейсные объекты и при необходимости реализуются вызовы их методов Release.

В модуле SysUtils объявлены две функции:

```
function Supports(const Instance: IUnknown;
const Intf: TGUID; out Inst): Boolean;
function Supports(Instance: TObject;
const Intf: TGUID; out Inst): Boolean;
```

Эти функции по ссылке Instance на интерфейс или интерфейсный объект и GUID интерфейса определяют, поддерживает ли Instance интерфейс Intf и, если да, возвращают в параметре Inst ссылку на реализующий его объект и True в качестве результата.

Фабрика классов и автоматический запуск сервера

Фабрика классов — это специальный интерфейс `IClassFactory`, экземпляр компонентного класса которого автоматически создается в момент старта сервера. Назначение этого интерфейса — дать клиенту возможность обращения к его методу `QueryInterface`, чтобы клиент мог получить ссылку на нужную службу сервера. Создание интерфейсного экземпляра фабрики классов обычно осуществляется в секции **initialization** соответствующего модуля.

Для автоматического старта сервера его необходимо зарегистрировать в системном реестре Windows. При регистрации в секции `HKEY_CLASSES_ROOT` реестра создается узел с именем сервера, в который помещается IID фабрики классов (рис. 25.1).

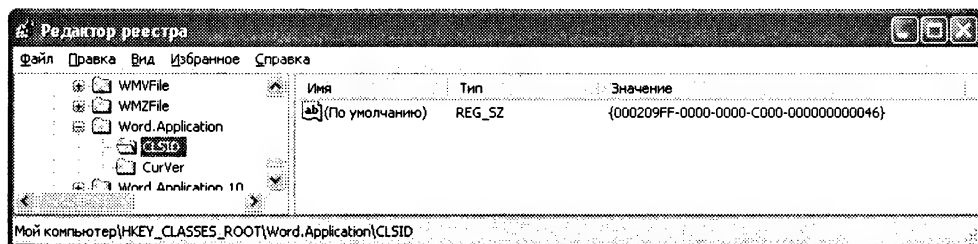


Рис. 25.1. Регистрация сервера Word.Basic

С помощью этой записи символьное имя сервера связывается с IID его фабрики классов. Чтобы связать IID с маршрутом доступа к серверу, в узле `HKEY_CLASSES_ROOT.CLSID` создается узел с именем IID, в подузел которого `Local-Server32` (или `InProcServer32` — зависит от реализации сервера — EXE или DLL) помещается маршрут (рис. 25.2).

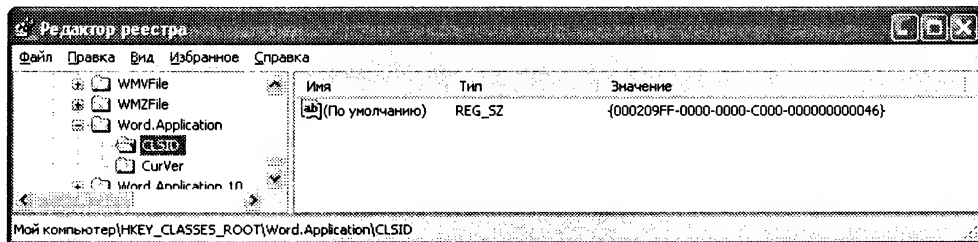


Рис. 25.2. Регистрация маршрута доступа к серверу

Для регистрации нового сервера программисту совсем не обязательно записывать вручную эти данные — Delphi имеет собственные средства регистрации сервера и удаления информации о нем из реестра. Для запуска сервера клиент обращается к системной функции `CoCreateInstance`, передавая ей в качестве параметра IID фабрики класса. Функция отыскивает в системном реестре нужный узел, считывает в нем маршрут доступа к серверу, загружает и запускает сервер, после чего обращается к методу `CreateInstance` фабрики класса для по-

лучения ссылки на нужный интерфейс. Замечу, что клиент, созданный с помощью Delphi, обычно имеет библиотеку типов сервера (см. ниже), с помощью которой он может обратиться непосредственно к методам интерфейса фабрики классов (без вызова `CoCreateInstance`).

Библиотека типов

Библиотека типов создается в Delphi автоматически в момент создания сервера COM. Ее назначение — дать клиенту исчерпывающую информацию обо всех интерфейсах сервера. Получив библиотеку типов, клиент может ссылаться на любые методы и свойства любого интерфейса, если в его предложение **uses** включена ссылка на эту библиотеку. Это дает возможность компилятору реализовать статическое (раннее) связывание с методами (свойствами) интерфейсного объекта, а также проконтролировать правильность обращения к ним. Библиотеки типов и специальный редактор для работы с ними будут подробно рассмотрены в обсуждаемых ниже примерах.

Если по каким-либо причинам у программиста, создающего приложение клиента, нет библиотеки типов сервера, он может вызывать свойства и методы по именам — такое связывание называется динамическим (поздним). Для реализации позднего связывания используется интерфейс диспетчеризации (наследник интерфейса `IDispatch`), который автоматически включается в библиотеку типов сервера COM. Позднее связывание характеризуется некоторыми дополнительными накладными расходами и обычно в практике создания клиентов в Delphi не используется.

Пример использования

В обсуждаемом ниже примере клиент обращается к серверу с указанием имени одного из графических файлов. В ответ сервер посылает ему этот файл, а клиент воспроизводит полученное изображение в своем компоненте `TImage`. Вид окна работающего клиента показан на рис. 25.3.

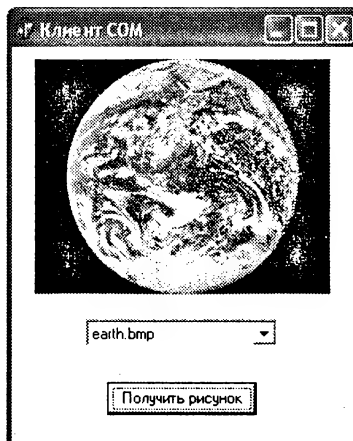


Рис. 25.3. Вид окна работающего клиента

Окно сервера в нашем примере не видно по той простой причине, что после завершения обслуживания клиента оно тут же исчезает с экрана.

Перед тем как начать работу над сервером, необходимо настроить среду Delphi: выберите команду Tools ► Environment Options и на вкладке Type Library открывшегося диалогового окна выберите язык Pascal (по умолчанию выбран язык IDL для совместимости с технологией CORBA).

Сервер

Начните новый проект (Chap_25\Example COM\ServBMP.dpr), выбрав команду File ► New ► Other и на вкладке ActiveX окна хранилища объектов дважды щелкнув на значке Automation Object. Запустится мастер создания объектов автоматизации (рис. 25.4).

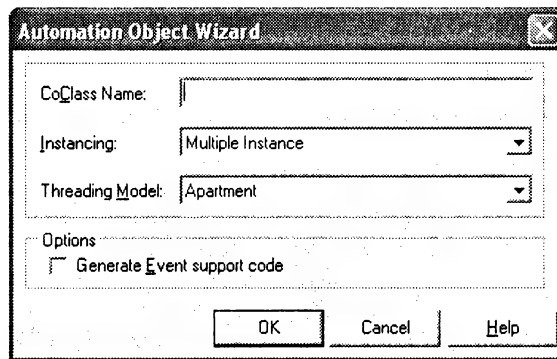


Рис. 25.4. Окно мастера создания объектов автоматизации

В поле CoClass Name нужно указать имя компонентного класса, в котором будет реализован интерфейс объекта автоматизации — введите в ней имя ServerBMP. Назначение остальных интерфейсных элементов описано ниже.

- **Instancing** — позволяет выбрать количество создаваемых объектов:
 - ☐ **Internal** — объект реализуется в виде DLL;
 - ☐ **Single** — объект существует в единственном экземпляре, повторный запрос клиента будет обслуживаться только после завершения работы с текущим клиентом;
 - ☐ **Multiple Instance** — каждый запрос клиента порождает новую копию объекта, которая может работать независимо от других.
- **Thread Model** — определяет используемую объектом модель потоков команд:
 - ☐ **Single** — каждый запрос клиентов к любому объекту использует один и тот же поток, поэтому запросы обрабатываются по очереди;
 - ☐ **Apartment** — для обслуживания каждого запроса создается свой поток, в котором выполняется одна копия объекта; обращения к общей глобальной памяти должны проходить при блокировке других потоков, в так называемых критических секциях; в этом режиме разрешается параллельное обслуживание запросов к *разным* объектам сервера (сервер может содержать более одного объекта COM);

- ☐ Free — разрешает параллельное обслуживание любых запросов;
- ☐ Both — совмещает модели Apartment и Free.
- ✱ Generate Event support code — при установке флажка мастер создаст код для обработки происходящих на сервере событий.

Оставьте умалчиваемые значения в списках Instancing и Threading Model, не трогайте флажок Generate Event support code и закройте окно щелчком на кнопке ОК. Откроется окно редактора библиотек типов (рис. 25.5).

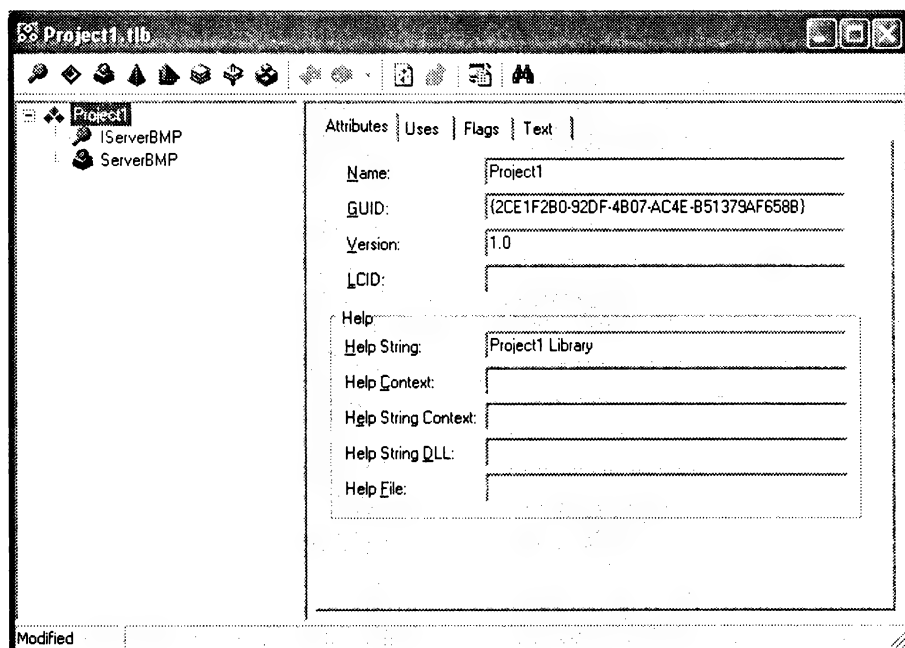


Рис. 25.5. Окно редактора библиотеки типов

С помощью редактора библиотек типов в создаваемый интерфейс объекта COM вносятся необходимые изменения. В частности, с помощью кнопки **New Method**  в интерфейс вставляется новый метод. На рис. 25.5 эта кнопка недоступна, так как в окне браузера не выбран интерфейс. Выберите интерфейс **IServerBMP**, щелкните на кнопке **New Method** и измените умалчиваемое имя метода **Method1** именем **GetBMP**. Вновь созданный интерфейсный метод должен выполнять всю работу по запросу клиента: он принимает в качестве входного параметра имя графического файла и возвращает клиенту массив считанных из файла байтов. Таким образом, метод является функцией с входным параметром типа **WideString** и типом результата **OleVariant**. Перейдите на вкладку **Parameters**, в раскрывающемся списке **Return Type** выберите тип возвращаемого результата **OleVariant** и сформируйте параметр обращения **Name** типа **WideString**¹ (рис. 25.6).

¹ Если в списке типов не найдены **OleVariant** или **WideString**, выполните описанную в начале параграфа процедуру настройки языковой поддержки библиотеки типов.

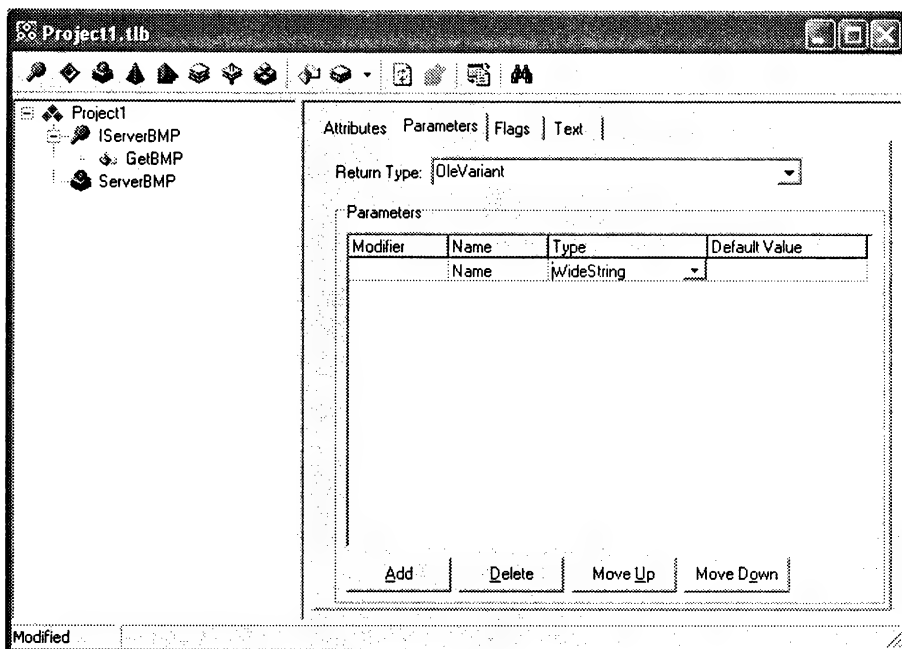


Рис. 25.6. Формирование возвращаемого методом GetBMP результата и входного параметра

Если теперь щелкнуть на кнопке Refresh  и перейти на вкладку Text, можно увидеть такое объявление интерфейсного метода:

```
function GetBMP (Name: WideString): OleVariant
    [dispid 00000001];
safecall;
```

Закройте окно редактора библиотек, нажав на клавишу F12. Вы увидите, что к проекту присоединены два новых модуля: Unit2 и Project1_TLB. Первый содержит объявление компонентного класса TServerBMP с заготовкой для функции GetBMP, а второй — обширный текст определения интерфейса IServerBMP и дублирующего его интерфейса диспетчеризации IServerBMPDisp.

Полезно с ним ознакомиться. В начале текста помещено следующее предупреждение (в переводе):

```
// ПРЕДУПРЕЖДЕНИЕ
// -----
// Объявленные в этом файле типы сгенерированы на основе
// данных из библиотеки типов. Если эта библиотека типов
// будет явно или неявно (путем импорта другой библиотеки
// типов, которая ссылается на данную) импортирована
// повторно или если во время работы с ней в редакторе
// библиотек будет нажата кнопка Refresh, то содержимое
// этого файла генерируется повторно и все модификации,
// выполненные вами вручную, будут утеряны.
```

Еще ниже по тексту вы увидите такой фрагмент:

```
//*****//
// Опережающее описание типов, определенных
// в библиотеке типов
//*****//
IServerBMP = interface;
IServerBMPDisp = dispinterface;
```

Как видите, мастер создания объекта автоматизации создал интерфейс диспетчеризации. Это открывает доступ к нашему серверу из программ, созданных с помощью Visual Basic, Java Script, SmallTalk или других языков программирования, которые не могут использовать указатели и вынуждены обращаться к методам и свойствам сервера по имени.

Сохраните ваш проект на диске: Unit2 под именем ServIMPL (от *implementation*), Unit1 под именем ServMain, а сам проект под именем ServBMP.

Перейдите в модуль ServIMPL и напишите код метода GetBMP, показанный в листинге 25.1.

Листинг 25.1. Код метода GetBMP¹

```
function TServerBMP.GetBMP(const Name: WideString):
OleVariant;
var
    V: OleVariant;
    F: File;
    Size: Integer;
const
    BMP_Path =                // Умалчиваемый маршрут доступа
                        // к графическим файлам
    'C:\Program Files\Common Files\Borland '+'
    'Shared\Images\Splash\16Color\';
begin
    // Создаем файл для чтения изображения
    AssignFile(F, BMP_Path+String(Name));
    Reset(F, 1);
    // Получаем длину файла
    Size := FileSize(F);
    // Создаем вариантный массив и наполняем его
    V := VarArrayCreate([1, Size], VarByte);
    BlockRead(F, VarArrayLock(V)^, Size);
    CloseFile(F);
    VarArrayUnLock(V);
    Result := V
end;
```

¹ Все представленные в книге тексты программ можно загрузить с сайта издательства по адресу <http://www.piter.com/download> — Примеч. ред.

Заслуживают комментария два момента. С помощью функции `VarArrayCreate` с переменной `V` связывается одномерный массив байтов с нижней границей 1 и верхней, соответствующей длине файла в байтах (переменная `Size`). Затем в массив считываются все байты файла с помощью функции `BlockRead`. Чтобы получить доступ к телу массива, он блокируется обращением к функции `VarArrayLock`. Поскольку эта функция возвращает *указатель* на первый байт массива, за обращением ставится символ `^`, чтобы сообщить компилятору о том, что речь идет не об *адресе*, а о *содержимом* области памяти, расположенной по этому адресу.

Обратите внимание, как просто строка `WideString` приводится к типу **String**:

String (Name)

Замечу, что в Delphi версии 7 функции для работы с вариантами определены в модуле `Variants` — вставьте ссылку на этот модуль в предложении **uses**. Запустите сервер, чтобы среда Delphi автоматически зарегистрировала его в реестре Windows.

Клиент

Начните новый проект (`Chap_25\Example COM\ClientBMP.dpr`) и поместите на пустую форму высотой 336 и шириной 268 пикселей панель размерами 181×225, на нее — компонент `Image1`, установив в его свойство `Align` значение `alClient`, а в свойство `Center` — значение `True`. Под панелью поместите компонент `ComboBox1` и кнопку `Button1`. В свойство `Style` компонента `ComboBox1` поместите `csDropDownList` и в его свойстве `Items` напишите такой список графических файлов:

```
athena.bmp    chip.bmp      construc.bmp
earth.bmp     skyline.bmp  technlgy.bmp
```

Присоедините к проекту файл библиотеки типов сервера `ServBMP_TLB.pas`, сделайте ссылку на него в предложении **uses** модуля `Unit1` и напишите обработчик `Button1OnClick`, показанный в листинге 25.2

Листинг 25.2. Обработчик `Button1OnClick`

```
procedure TForm2.Button1Click(Sender: TObject);
var
    II, Server: IServerBMP;
    V: OleVariant;
    F: File;
    Size: Integer;
begin
    if ComboBox1.Text = '' then Exit;
    // Активизируем сервер
    II := coServerBMP.Create;
    // В переменную Server помещаем ссылку
    // на интерфейсный объект
    II.QueryInterface(IServerBMP, Server);
```

```
// Получаем массив с пикселями изображения
V := Server.GetBMP(WideString(ComboBox1.Text));
// Создаем временный файл
AssignFile(F, 'temp.bmp');
Rewrite(F, 1);
// Определяем длину файла
Size := VarArrayHighBound(V, 1);
// Записываем вариантный массив в файл
BlockWrite(F, VarArrayLock(V)^, Size);
CloseFile(F);
// Показываем изображение и уничтожаем файл
Image1.Picture.LoadFromFile('temp.bmp');
Erase(F)
end;
```

Хочу лишний раз обратить ваше внимание на обязательную ссылку на модуль библиотеки типов в предложении **uses**: без нее стали бы неизвестными для компилятора ссылки на интерфейс `IServerBMP`, его интерфейсный метод `GetBMP` и функцию класса `coServerBMP.Create`. Обращение к этой функции автоматически активизирует сервер (его окно на мгновение появится на экране), при этом создается компонентный объект фабрики класса, ссылку на которой мы разместили в переменной `II`. После этого обращением к методу `QueryInterface` объекта `II` в переменную `Server` помещается ссылка на компонентный объект класса `IServerBMP`, в результате чего, наконец, становится доступной нужная нам функция `GetBMP`. Заметьте, что объект фабрики класса `II` не имеет функции `GetBMP`, поэтому обращение `II.GetBMP` сразу после создания фабрики привело бы к ошибке. С другой стороны, присваивание ссылки на этот объект переменной типа `IServerBMP` не страшно, так как оба интерфейса порождены от общего родителя `IUnknown`, следовательно, имеют общий метод `QueryInterface`.

Использование вариантов в технологии OLE

Delphi располагает различными средствами поддержки OLE. Некоторые из них обеспечиваются возможностями вариантов, так как одним из значений варианта может быть OLE-объект.

Для создания OLE-объекта используется стандартная функция `CreateOleObject`:

```
function CreateOleObject(const ClassName: String):
    Idispatch;
```

Функция по заданному имени класса сервера `ClassName` возвращает так называемый идентификатор интерфейса `Idispatch`, который нужно использовать для обращения к серверу. Имя `ClassName` должно быть зарегистрировано в реестре Windows. Идентификатор интерфейса определяет интерфейс сервера, то есть набор допустимых методов OLE-объекта, которые реализует сервер. Имя класса

сервера и его интерфейс должны быть известны программисту. В следующем примере после щелчка на кнопке `bbRun` учебной программы создается OLE-объект для связи с текстовым редактором MS Word, и этому редактору передается текст, набранный в компоненте `mmOutput`.

```
{ $R *.DFM }
```

```
Uses ComObj;
```

```
procedure TfmExample.bbRunClick(Sender: TObject);
var
  OLE: Variant;
begin
  OLE := CreateOleObject('Word.Basic');
  OLE.FileNew;
  OLE.Insert(mmOutput.Text);
end;
```

Функция `CreateOleObject` определена в модуле `COMOBJ`, поэтому ссылку на этот модуль необходимо указать в самом начале исполняемого раздела модуля `Form1` (после строки `{ $R *.DFM }`).

Если вы используете технологию автоматизации OLE в одной из подпрограмм DLL, она должна ссылаться на модуль `ComObj`, в иницилирующей части которого создается объект компонентного класса для доступа к дуальным интерфейсам. Однако инициализирующие части модулей, на которые ссылается DLL, *не выполняются автоматически* в момент старта программы, если только они не упомянуты в предложении **uses** главной программы или любого ее модуля, поэтому в общем случае не будет создан компонентный объект модуля `COMOBJ` и обращение к `CreateOleObject` вызовет сообщение об ошибке. При разработке DLL для использования в среде Delphi эта проблема легко решается добавлением ссылки на модуль `COMOBJ` в главной программе или любом ее модуле. Однако если DLL разрабатывается для продажи или работы с другими системами программирования (Visual Basic, C++, JBuilder и т. п.), эта проблема не всегда разрешима, так как в инициализирующей части модуля используются обращения к полям и методам, закрытым для доступа вне модуля.

Замечу попутно, что если ваша программа не может правильно вызвать функцию `CreateOleObject`, в ряде случаев может помочь явный вызов функции `API CoInitialize`. Назначение функции — произвести подготовку (инициализацию) механизма COM. Эта функция не описана в модуле `Windows` и, следовательно, «не известна» Delphi. Ее прототип таков:

```
function CoInitialize(Reserved: Pointer): HRESULT;
stdcall; external 'OLE32.dll';
```

Единственный параметр обращения к этой функции зарезервирован и должен содержать **NIL**. Функция возвращает `S_OK` (тип `HRESULT` и возможные его значения описаны в модуле `Windows`), если обращение к ней прошло успешно.

Если вы запустите программу, то обнаружите, что на экране появляется окно MS Word, которое немедленно после этого исчезает¹. Происходит это потому, что

жизненный цикл активизации и работы OLE-сервера связан с жизненным циклом и содержимым переменной-варианта. В нашем примере переменная OLE определена в теле процедуры `bbRunClick`, то есть является локальной переменной. Локальные переменные создаются в момент входа в подпрограмму и уничтожаются в момент завершения ее работы — вот почему MS Word так быстро закрывается. Чтобы пример стал имитацией нормальной работы с MS Word, переделаем его: переменную OLE вынесем в раздел **public** класса `TfmExample` нашей программы, а на кнопке `bbRun` будем щелкать дважды — один раз для активизации MS Word, а второй — чтобы сохранить набранный в нем текст в текстовом файле, закрыть MS Word и прочитать содержимое файла в поле `mmOutput`.

```

type
  TfmExample = class (TForm)
  ...
  public
  { Public declarations }
    OLE: Variant;
end;
...
Uses ComObj;

procedure TfmExample.bbRunClick(Sender: TObject);
const
  FileName = 'c:\temp.txt';
begin
  if VarType(OLE) <> varDispatch then
    begin           // С вариантом не связан OLE-объект:
                   // запускаем MS Word
    OLE := CreateOleObject('Word.Basic');
    // Делаем его видимым
    OLE.AppShow;
    // Создаем новый документ
    OLE.FileNew;
  end else // VarType(OLE) = varDispatch
  {OLE-объект уже создан: вставляем в поле mmOutput текст из
  подготовленного файла и закрываем MS Word}
  begin
    // Сохраняем в файле только текст
    OLE.FileSaveAs(FileName, 3);
    // Закрываем документ
    OLE.FileClose(2);
    // Закрываем MS Word
    OLE := 0;

```

¹ Перед прогоном программы следует, во-первых, убедиться в том, что на вашем ПК установлен текстовый процессор MS Word, и, во-вторых, закрыть его, если он перед этим был открыт. Если MS Word был открыт перед прогоном программы, он не исчезнет с экрана.

```

// Проверяем существование файла
if FileExists (FileName) then
begin
  mmOutput.Lines.LoadFromFile (FileName) //Загружаем файл
  DeleteFile (FileName)                // и уничтожаем его
end
end
end;

```

После первого щелчка на кнопке `bbRun` в переменную OLE будет помещен интерфейс сервера, что позволяет программе сделать окно MS Word видимым и открыть в нем новый документ. Предполагается, что в этот момент пользователь должен перейти к работе с текстовым процессором и подготовить в нем произвольный текст, после чего он второй раз щелкает на кнопке `bbRun`. Поскольку переменная OLE является полем класса `TfmExample`, ее значение к этому моменту остается `Idispatch`, и обработчик `bbRunClick` выполняет вторую часть работы: сохраняет набранный в процессоре текст в файл, закрывает документ и сам сервер, поместив в переменную OLE новое значение.

Использование серверов пакета MS Office

Практика показывает, что программы пакета MS Office (Excel, Word, Power Point и т. п.) устанавливаются на подавляющем большинстве компьютеров, работающих под управлением 32-разрядных версий Windows. Каждая из этих программ является COM-сервером, и, следовательно, любой входящий в нее объект может использоваться вашей программой как свой собственный.

Как уже говорилось выше, существуют два способа обращения к методам и свойствам COM-объекта: путем ссылки на его библиотеку типов (раннее связывание) и по имени (позднее связывание). Для Delphi предпочтительным является раннее связывание, так как в этом случае компилятор может проконтролировать правильность обращения к свойствам и методам внешних объектов, а создаваемый им код исполняется, в общем случае, быстрее. С другой стороны, базовый язык обращения к серверам MS Office — Visual Basic for Application (VBA) не поддерживает работу с указателями и, следовательно, не может использовать интерфейсы. Специально для такого рода языков (помимо VBA с указателями не работают также языки Java, JavaScript, SmallTalk и некоторые другие) в технологию COM введены интерфейсы диспетчеризации, позволяющие обращаться к методам и свойствам по имени. При установке MS Office можно установить справки по VBA, в которых детально описываются интерфейсные службы серверов MS Office с указанием назначения методов и свойств и параметров обращения к ним. Фактически это единственные доступные программисту документы, на которые он вынужден опираться при программировании доступа к мощным возможностям серверов MS Office. Замечу, что при стандартной установке пакета справки по VBA на диске не размещаются. Если в каталоге `Program Files\Microsoft Office\Office` вы не найдете файлов `vbaxlX.hlp` (справка по Excel), `vbawrdX.hlp` (справка по Word) и т. п. (X — номер версии MS Office: для Office 97 — 8, для Office 2000 — 9, для Office XP — 10), вы долж-

них их добавить с помощью апплета Установка и удаление программ панели управления (Пуск ► Настройка ► Панель управления).

В Delphi имеются компоненты вкладки Servers, позволяющие обращаться к COM-объектам серверов с помощью библиотек типов, однако эти компоненты практически не документированы. Во всех случаях изучение обширных текстов библиотек (например, библиотека Excel_TLB.pas для Office 2000 содержит почти 40 000 строк¹) мало что дает даже опытному программисту.

В этом разделе приводится краткое описание основных объектов двух наиболее популярных серверов — Excel и Word. Поскольку специально для версии Office 97 язык VBA был существенно расширен, этот материал нельзя использовать для работы с более ранними версиями пакета.

Основные объекты серверов Excel и Word

В терминологии VBA используются понятия объект и коллекция. Объект — это обычный интерфейсный объект COM, имеющий свойства, методы и события. Коллекция — это группа однотипных объектов. Например, главный объект сервера Excel — Application — определяет основные свойства и методы сервера, коллекция Worksheets — это набор табличных страниц в текущей рабочей книге и т. д. Представленные ниже иерархии объектов и коллекций взяты из файлов vbaXXX.hlp (для Office 97). В отличие от объектов VCL они построены не по принципу наследования, а по функциональной подчиненности.

Объекты Excel

Сервер Excel — это мощный табличный процессор, реализующий размещение и обработку различного рода данных (как числовых, так и текстовых), в том числе — построение на их основе графиков и диаграмм. При работе с Excel создается так называемая рабочая книга (файл данных) с одним или несколькими листами. Все листы одной рабочей книги могут быть связаны друг с другом, что позволяет организовать совместные вычисления над размещенными на них данными.

На рис. 25.7 представлена функциональная структура объектов и коллекций Excel.

Объект Application имеет многочисленные свойства, методы и события, управляющие сервером в целом. Только с его помощью, например, можно визуализировать полнофункциональное окно текстового процессора. Его центральное свойство Workbooks открывает доступ ко всем открытым в процессоре рабочим книгам.

У каждой рабочей книги есть свойства Worksheets и Charts, представляющие собой коллекции листов и диаграмм. Первоначально коллекция Workbooks пуста. Чтобы создать хотя бы одну рабочую книгу, нужно обратиться к методу Workbook.Add, который создает рабочую книгу с количеством пустых листов, определяемым значением свойства Application.SheetsInNewWorkbook. У каждого рабочего листа есть свойство Cells (I, J), определяющее содержимое ячейки, лежащей на пересечении I-й строки с J-м столбцом (нумерация строк и столб-

¹ Для импортирования библиотеки типов любого зарегистрированного на вашей машине COM-сервера используется диалоговое окно Import Type Library (открывается командой Project ► Import Type Library): выберите нужный сервер в списке и щелкните на кнопке Create Unit.

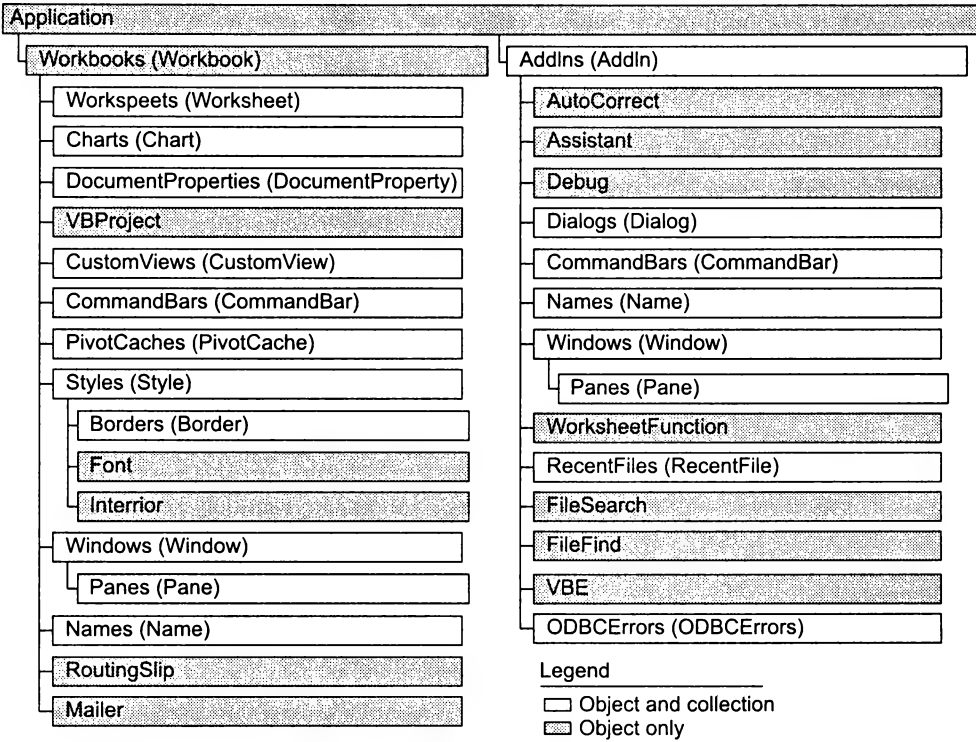


Рис. 25.7. Функциональная структура объектов и коллекций Excel

цов начинается с 1). Если при обращении к Cells номера столбца и строки опущены, считается, что речь идет о текущем диапазоне ячеек, заданным значением свойства Worksheets.Range. Если необходимо изменить умалчиваемые свойства столбца или строки, используются объекты Worksheets.Columns и Worksheets.Rows. Помимо рабочих листов с рабочей книгой связывается объект Charts, представляющий собой коллекцию диаграмм. С каждой диаграммой связывается объект SeriesCollection, хранящий данные, по которым строится диаграмма.

Объекты Word

Текстовый процессор Word является популярнейшим средством создания и оформления (форматирования) текстовых документов. С его помощью, например, была создана и оформлена эта книга, как, впрочем, и все другие мои книги и книги многих известных мне авторов. При работе с Word фундаментальными понятиями являются документ, абзац и стиль. Документ определяет файл данных. Абзац — это совокупность символов, обрамленная служебными символами конца строк, разрыва колонки или разрыва раздела. Наконец, стиль — это совокупность признаков оформления текста: его шрифт, положение на странице, выравнивание и т. п. Стиль — неперемный атрибут каждого абзаца, то есть изменение стиля абзаца приводит к его переформатированию. Однако стиль может изменяться внутри аб-

заца — для выделения группы символов шрифтом, цветом символов и/или фона и т. п.

На рис. 25.8 показана функциональная иерархия объектов Word.

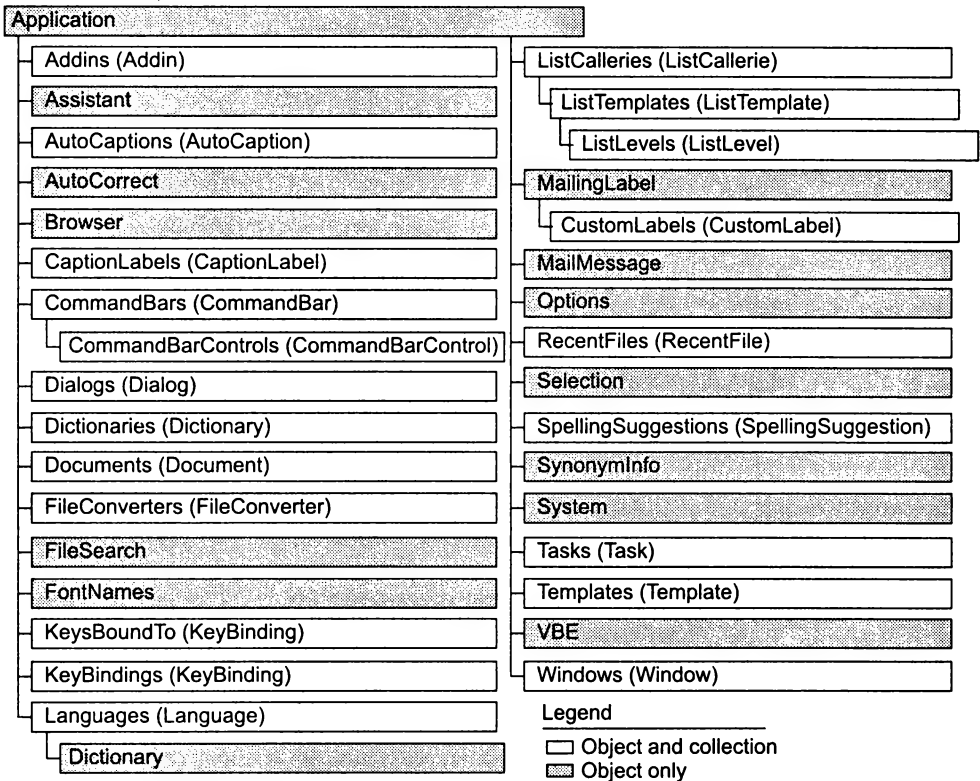


Рис. 25.8. Функциональная структура объектов и коллекций Word

Центральный объект `Application` имеет такое же назначение, что и одноименный объект `Excel`, — он определяет свойства, методы и события на уровне всего сервера. Его свойство `Documents` представляет собой коллекцию открытых документов. С помощью метода `Open` этого объекта можно открыть ранее созданный документ, а методом `Add` — создать новый документ, основанный на шаблоне `Normal.dot`. Каждый документ имеет коллекцию абзацев `Paragraphs`. С помощью методов этого объекта `Add`, `InsertParagraph`, `InsertParagraphAfter`, `InsertParagraphBefore` можно вставить новый абзац в уже существующий текст или добавить абзац в конец документа. В свою очередь, каждый абзац имеет многочисленные свойства, позволяющие нужным образом отформатировать текст. Как и в `Excel`, важную роль в иерархии объектов `Word` играет объект `Range`, определяющий диапазон абзацев. Свойство `Text` этого объекта содержит текст диапазона.

Глава 26

Создание компонентов с помощью ModelMaker

Как уже упоминалось (см. введение к книге), инструмент автоматизированного программирования CASE (Computing Assisted Software Engineering) под названием ModelMaker берет на себя значительную часть «рутинной» работы по кодированию программы, позволяя программисту сосредоточиться на творческой стороне этого процесса. Эта утилита разработана голландской фирмой ModelMaker Tools и, как утверждается в документации, может работать совместно с любой версией Delphi (в том числе — с первой). Утилита не является неотъемлемой составной частью Delphi и устанавливается по желанию пользователя. Для установки вставьте дистрибутивный компакт-диск Delphi в дисковод вашего компьютера и выберите продолжение ModelMaker 6.20.

Главное назначение ModelMaker — автоматизировать процесс создания классов. Подобно другим средствам CASE, ModelMaker может создавать и поддерживать диаграммы, отражающие связи между классами и их членами. Однако в отличие от других средств CASE (а эти средства, в основном, предназначены для автоматизированной разработки структур баз данных, оптимизации реляционных связей между таблицами и т. п.), ModelMaker тесно связан с языком и средой Delphi.

Новые классы создаются в сложных программных проектах для оптимизации повторного использования раз созданного кода, а также при создании новых компонентов. Поэтому в этой главе, иллюстрирующей методiku работы с ModelMaker, мы также коснемся общей проблемы создания новых компонентов.

Методика создания компонентов

В этом разделе рассматриваются разнообразные аспекты, связанные с созданием собственных (нестандартных) компонентов. Если быть более точным, в ней рассматривается методика создания новых классов, так как компонент является лишь вариантом реализации объекта класса, лежащего в его основе.

Зачем нужны новые компоненты

В состав Delphi входит более 400 компонентов, рассчитанных на самые разные сферы применения. И, тем не менее, в некоторых случаях приходится создавать новые компоненты, поскольку:

- в существующих нет нужной функциональности, а новый компонент не только предоставит эту функциональность, но и будет использоваться много раз в различных приложениях;
- вы желаете деструктурировать сложный алгоритм на несколько относительно простых и устойчивых к ошибкам фрагментов, используя для реализации каждого один или несколько классов;
- вы хотите создать компонент на продажу (стандартная цена нового компонента — от 10 до 100 долларов; часто такие деньги позволяют покупателям сэкономить время на его создание, которое в стоимостном выражении намного больше этих затрат, так что такого рода бизнес может приносить неплохой доход).

Как бы там ни было, вы читаете этот раздел, а значит, у вас есть свое мнение по этому вопросу. Приготовьтесь! Процесс создания нового компонента потребует от вас основательного владения техникой объектно-ориентированного программирования.

Этапы разработки

В этом разделе перечисляются этапы разработки компонента, кроме, возможно, самого главного: этапа «вынашивания» основной идеи компонента, то есть решение для себя вопроса, — чем будет отличаться новый компонент от существующих и будет ли он полезен в других программах или другим программистам. Последнее обстоятельство — возможное тиражирование компонента — очень важно, так как если компонент используется всего один раз или в одной-единственной программе, вряд ли стоит тратить дополнительные усилия на придание ему функциональности именно компонента, скорее всего, задачу с успехом решит специализированный модуль.

Выбор родительского класса

Огромную роль играет правильный выбор класса, от которого новый компонент будет наследовать свои основные свойства, методы и события. На рис. 26.1 представлена иерархия важнейших классов Delphi, на основе которых создано подавляющее большинство компонентов.

Родоначалник иерархии TObject может использоваться как предок только при создании невидимого компонента (точнее, класса, так как компонентами в Delphi принято считать прямых или косвенных потомков TComponent). Функциональность TObject связана, в основном, с поддержкой механизма распределения в динамической памяти объектов VCL, предоставлении программе информации о типе объекта (Run Time Type Information, RTTI), с поддержкой интерфейсов и т. п. Использовать его в качестве родительского класса можно для относительно простых компонентов, не требующих сохранения их на диске (то есть создаваемых не на этапе конструирования, а в ходе прогона программы) и не связанных

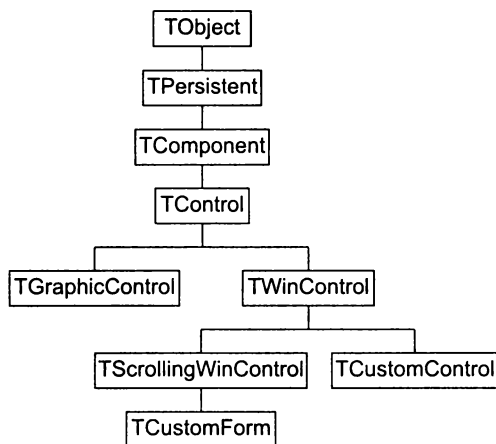


Рис. 26.1. Иерархия важнейших классов Delphi

с объектами других классов. Типичным примером наследника `TObject` может быть компонент `TIniFile`, который инкапсулирует в себе все необходимое для доступа к файлам инициализации Windows.

Класс `TPersistent` передает потомкам умение работать с потоками данных, чтобы читать из них (записывать в них) свойства связанных с классом других объектов. Например, сложное свойство `TMemo.Lines` относится к прямому потомку `TPersistent` — классу `TStrings`; к этому же классу относится свойство `Items` компонента `TListBox`, поэтому с помощью метода `Assign` класса `TPersistent` можно содержимое одного компонента передать другому:

```
Memor.Lines.Assign(ListBox1.Items);
```

Вряд ли вам будет достаточно только этой функциональности, поэтому на практике в качестве родительского класса невидимых компонентов обычно используется `TComponent`, наследники которого являются полноценными *компонентами* — они могут регистрироваться в палитре компонентов Delphi, к их опубликованным свойствам открыт доступ на этапе конструирования, они могут владеть и управлять другими компонентами, при необходимости они могут быть преобразованы в компоненты `ActiveX`.

Класс `TControl` является базовым для всех визуальных компонентов. Именно в нем инкапсулируются все события, свойства и методы, общие для любого визуального компонента. Однако вряд ли следует выбирать этот класс в качестве родительского: его ближайшие наследники `TGraphicControl` и `TWinControl` разделяют все визуальные компоненты на две группы: имеющие дефицитный системный ресурс — дескриптор окна (потомки `TWinControl`) и не имеющие его (потомки `TGraphicControl`). Класс `TCustomControl` передает своим потомкам механизм канвы.

Таким образом, для создания невидимого компонента следует наследовать его основные свойства от класса `TComponent`, для создания визуального — от класса `TGraphicControl`, `TWinControl` или `TCustomControl`, в зависимости от того, нужен ли вашему компоненту дескриптор окна и свойство `Canvas`.

Эти рекомендации носят самый общий характер. В реальном случае следует выбирать потомка одного из классов, который бы в максимальной степени приближался в функциональном плане к вновь создаваемому компоненту: только в этом случае вы сможете с максимальной эффективностью воспользоваться преимуществами ООП, в частности, повторным использованием ранее созданного кода. Замечу, что именно для этих целей разработчики Delphi предусмотрели множество промежуточных классов с именами типа `TCustomNNNN`, в которых инкапсулируются наиболее общие для некоторой функциональности свойства, методы и события. Например, стандартный компонент `TLabel` порожден от класса `TCustomLabel`, являющегося базовым для любых визуальных компонентов, которые демонстрируют на экране текст, но не имеют дескриптора окна. Класс `TCustomStaticText` обладает почти такой же функциональностью, но получает дескриптор окна и так далее. Обычно в программе экземпляры этих классов не используются, так как в большинстве случаев они (классы) являются абстрактными. Однако если вы создаете собственную «неоконную» метку, новый класс, возможно, окажется удобнее наследовать от `TCustomLabel`, а не от `TLabel`, точно так же как наследование от `TCustomStaticText` будет удобнее, чем от `TStaticText`, при создании «оконной» метки и так далее.

Добавление свойств

Напомню, что только те свойства, которые включены в секцию **published**, появляются в окне инспектора объектов, и программист может их изменять на этапе конструирования. В этом пункте речь пойдет именно об опубликованных свойствах, так как вид их представления в инспекторе объектов существенно зависит от типа свойства.

ПРИМЕЧАНИЕ Все приводимые в этом разделе фрагменты программного кода носят иллюстрирующий характер и не имеют самостоятельного значения.

Простые свойства. Простыми являются свойства, имеющие значение одного из «простых» типов — целого, вещественного, строкового, логического и т. п. Такие свойства не требуют специальных механизмов доступа. Вот как, например, в компонент можно добавить два простых свойства:

type

```
TMyComp = class(TWinControl)
```

```
private
```

```
    // Закрытые поля для свойств
```

```
    FIntegerProp: Integer;
```

```
    FStringProp: String;
```

```
published
```

```
    // Простые свойства
```

```
    property IntegerProp:
```

```
        Integer read FIntegerProp write FIntegerProp;
```

```
    property StringProp:
```

```
        String read FStringProp write FStringProp;
```

```
end;
```

Замечу, что обычно все поля для свойств именуются с префиксом F и размещаются в секции **private**.

Перечисленные свойства. Особенностью перечисленных свойств является predetermined набор их возможных значений. Поэтому в окне инспектора объектов в поле значения свойства автоматически появляется кнопка, щелчок на которой приводит к раскрытию списка возможных значений. Добавим в компонент перечисленное свойство:

```
type
  TEnumProp = (epZero, epOne, epTwo, epThree);
  TMyComp = class(TWinControl)
  private
    // Закрытые поля для свойств
    ...
    FEnumProp: TEnumProp;
  published
    ...
    // Перечисленное свойство
    property EnumProp: TEnumProp read FEnumProp
      write FEnumProp;
  end;
```

Замечу, что свойства логического типа являются, по существу, перечисленными, поэтому их значения в окне инспектора объектов тоже можно выбирать в списке.

Свойства-множества. Свойство-множество в окне инспектора объектов раскрывается в список логических свойств. Вот как можно его добавить:

```
type
  TSetPropItems = (spA, spB, spC, spD);
  TSetProp = set of TSetPropItems;
  TMyComp = class(TWinControl)
  private
    // Закрытые поля для свойств
    ...
    FSetProp: TSetProp;
  published
    ...
    // Свойство-множество
    property SetProp: TSetProp read FSetProp
      write FSetProp;
  end;
```

Свойства-объекты. Как известно, полями компонента могут быть объекты других классов. Это позволяет вставлять в компонент свойства-объекты. При этом необходимо учесть три аспекта:

- ☛ после создания компонента нужно создать объекты-поля;
- ☛ перед удалением компонента нужно удалить объекты-поля;
- ☛ установка нового значения свойства (то есть присваивание свойству другого объекта) должна сопровождаться освобождением ранее созданного.

Пусть в компонент добавляется свойство-объект `ObjProp`, которое экспонирует два своих простых свойства `Prop1` и `Prop2` (эти свойства будут доступны в окне инспектора объектов). В свойство-объект помещается объект следующего класса:

```
type
  TInnerObject = class (TPersistent)
  private
    FProp1: Integer;
    FProp2: String;
  published
    property Prop1: Integer read FProp1 write FProp1;
    property Prop2: String read FProp2 write FProp2;
end;
```

Для корректного создания/уничтожения компонента следует перекрыть конструктор и деструктор компонента (`FObjProp` — поле, содержащее внутренний объект):

```
constructor TMyComp.Create (AOwner: TComponent);
// При создании объекта создается внутренний
// объект-свойство
begin
  inherited;
  FObjProp := TInnerObject.Create;
end;
```

```
destructor TMyComp.Destroy;
// При удалении вначале удаляется внутренний
// объект-свойство
begin
  FreeAndNil (FObjProp);
  inherited;
end;
```

Объявим свойство простейшим образом:

```
TMyComp = class (TWinControl)
  private
    // Закрытые поля для свойств
    ...
    FObjProp: TInnerObject;
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
  published
    ...
    // Свойство-объект
    property ObjProp: TInnerObject read FObjProp
      write FObjProp;
end;
```


Тогда при замене значения свойства старый объект не будет удален, так как поле `FObjProp` хранит лишь *указатель* на динамически распределенный объект, и простая замена указателей не ведет к освобождению памяти. Поэтому устанавливать новое значение нужно в два этапа: сначала удалить старый объект и только затем заменить указатель:

```
TMyComp = class (TWinControl)
private
    ...
    FObjProp: TInnerObject;
    procedure SetObjProp(const Value: TInnerObject);
public
    ...
published
    property ObjProp:
        TInnerObject read FObjProp write SetObjProp;
end;
procedure TMyComp.SetObjProp(const Value: TInnerObject);
begin
    if Assigned(Value) then
        begin
            FreeAndNil (FObjProp);
            FObjProp := Value
        end
    end;
```

Свойства-массивы. Такого рода свойства содержат сразу несколько значений, доступ к каждому из которых возможен с помощью индекса. Например, свойство `Lines` компонента `TMemo` содержит индексированный массив строк:

```
Label1.Caption := Memo1.Lines[0];
```

В отличие от обычных массивов, свойства-массивы могут иметь индексные выражения строкового типа. Например, у компонента `TTable` свойство `FieldValues` содержит массив значений текущей записи, который индексирован по именам полей:

```
Table1.FieldValues['BName'] := 'Настольная книга программиста';
```

Важной особенностью свойств-массивов является то, что, как правило, они не имеют соответствующего поля для хранения значений, а доступ к их значениям реализуется методами `SetXXXX` и `GetXXXX`.

Добавим в компонент `TMyComp` два свойства-массива. Одно из них (`DayOfWeek`) будет возвращать название дня недели по его номеру, другое (`DayNumber`) — номер дня недели по его названию (листинг 26.1).

Листинг 26.1. Определение свойств-массивов¹

```
type TMyComp = class (TWinControl)
```

¹ Все представленные в книге тексты программ можно загрузить с сайта издательства по адресу <http://www.piter.com/download> — *Примеч. ред.*

```
private
...
function GetDayName(const AIndex: Byte): String;
function GetDayNumber(const AName: String): Byte;
public
...
// Свойства-массивы нельзя публиковать в секции
// published!
property DayOfWeek[const AIndex: Byte]:
    String read GetDayName;
property DayNumber[const AName: String]:
    Byte read GetDayNumber;
published
...
end;

procedure Register;

implementation
...

// Массив названий дней недели:
const
    Names: array [1..7] of String[11] = ('понедельник',
        'вторник', 'среда', 'четверг', 'пятница', 'суббота',
        'воскресенье');

function TVFStrangeComp.GetDayName(const AIndex: Byte):
    String;
begin
    if AIndex in [1..7] then
        Result := Names[AIndex]
    else
        raise Exception.Create(
            'Неверный номер дня недели: '+IntToStr(AIndex))
end;

function TVFStrangeComp.GetDayNumber(const AName: String):
    Byte;
begin
    Result := 0;
    repeat
        inc(Result)
    until (Result=8) or (UpperCase(Names[Result]) =
        UpperCase(AName));
    if Result=8 then
```

```

    raise Exception.Create('Неверное имя дня недели: '+
        AName)
end;

```

Обратите внимание: свойства-массивы нельзя объявлять в секции **published**, то есть они не могут быть доступны на стадии конструирования. Это ограничение в ряде случаев можно обойти, объявив свойства скалярными и создав специализированные редакторы свойств.

Умалчиваемые значения свойств. Если компонент TMyComp поместить на форму, он будет невидимой точкой, так как все его поля в момент создания обнуляются, поэтому свойства Height и Width, определяющие видимые размеры компонента, будут равны 0. Присваивание умалчиваемых значений свойствам компонента реализуется в его конструкторе:

```

constructor TMyComp.Create(AOwner: TComponent);
// При создании объекта осуществляется присваивание
// умалчиваемых значений свойствам и создается внутренний
// объект-свойство
begin
    inherited;
    Width := 20;
    Height := 20;
    FObjProp := TInnerObject.Create;
end;

```

Определение любого свойства можно сопровождать зарезервированным словом **default**:

```

property IntegerProp:
    Integer read FIntegerProp write FIntegerProp default 100;

```

Однако такое определение отнюдь не означает определение умалчиваемого значения (английское слово *default* означает *неявка*, *невыполнение*; в компьютерной литературе часто используется как синоним слову *умалчиваемый*). Оно лишь указывает, что если значение свойства равно 100, оно не будет сохраняться в DFM-файле формы и считываться из него при повторной загрузке, так как оно устанавливается автоматически *перед чтением* данных. Если значение отлично от определенного директивой **default**, оно сохраняется в файле и считывается из него, заменяя предварительно установленное значение. Рекомендуется по возможности чаще использовать директиву **default**, так как она сокращает время загрузки/сохранения формы.

Умалчиваемое свойство-массив. Директива **default**, стоящая в определении свойства-массива, делает это свойство умалчиваемым. Это означает, что доступ к свойству может проходить без ссылки на свойство. Пусть, например, сделано умалчиваемым свойство TMyComp.DayOfWeek:

```

property DayOfWeek[const AIndex: Byte]:
    String read GetDayName default;

```

Тогда допустим такой оператор:

```

Label1.Caption := MyComp[2]; // 'вторник'

```

В компоненте может быть только одно умалчиваемое свойство-массив.

Доступ к свойствам внутренних компонентов. Во многих случаях вновь создаваемый компонент включает в себя другие компоненты — метки, поля, таймеры и т. п. Некоторые свойства таких компонентов обычно требуется включить в опубликованные свойства вновь создаваемого, чтобы иметь к ним доступ на этапе конструирования. Для этого части **read** и **write** свойства должны ссылаться на методы, в которых осуществляется доступ к соответствующим свойствам внутренних компонентов.

Вставим в компонент TMyComp таймер TTimer и опубликуем его свойства Enabled и Interval так, как показано в листинге 26.2.

Листинг 26.2. Доступ к свойствам внутренних компонентов

```
TMyComp = class(TWinControl)
  private
    ...
    FTimer: TTimer;
    function GetTimerEnabled: Boolean;
    function GetTimerInterval: Cardinal;
    procedure SetTimerEnabled(const Value: Boolean);
    procedure SetTimerInterval(const Value: Cardinal);
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    ...
  published
    ...
    // Доступ к свойствам таймера
    property TimerEnabled: Boolean
      read GetTimerEnabled write SetTimerEnabled;
    property TimerInterval: Cardinal
      read GetTimerInterval write SetTimerInterval;
  end;

  ...

  constructor TMyComp.Create(AOwner: TComponent);
  // При создании объекта осуществляется присваивание
  // умалчиваемых значений свойствам, а также создаются
  // внутренние объекты
begin
  inherited;
  Width := 20;
  Height := 20;
  FObjProp := TInnerObject.Create;
  FTimer := TTimer.Create(Self);
end;
```

```

destructor TMyComp.Destroy;
// При удалении вначале уничтожаются внутренние объекты
begin
    FObjProp.Free;
    FTimer.Free;
    inherited;
end;

function TMyComp.GetTimerEnabled: Boolean;
begin
    Result := FTimer.Enabled
end;

function TMyComp.GetTimerInterval: Cardinal;
begin
    Result := FTimer.Interval
end;

procedure TMyComp.SetTimerEnabled(const Value: Boolean);
begin
    FTimer.Enabled := Value
end;

procedure TMyComp.SetTimerInterval(const Value: Cardinal);
begin
    FTimer.Interval := Value
end;

```

Публикация свойств родительского класса. В некоторых случаях может понадобиться публикация неопубликованных свойств родительского класса. Если, например, компонент TMyComp будет получать фокус ввода, ему понадобится опубликовать свойства TabOrder и TabStop своего предка TWinControl, у которого эти свойства объявлены в секции **public** и, таким образом, недоступны на этапе конструирования.

Публикация заключается в простом *упоминании* (не объявлении!) нужного свойства в секции **published**:

```

TMyComp = class (TWinControl)
...
published
...
    property TabOrder;
    property TabStop;
end;

```

Нельзя опубликовать закрытые свойства родителя (то есть объявленные в секции **private**).

Создание методов

Создание методов, как правило, не вызывает особых трудностей. Следует отметить три момента.

- При перекрытии конструктора его новое объявление должно сопровождаться директивой **override**, если перекрываемый конструктор родительского класса объявлен как виртуальный (все потомки класса TComponent имеют виртуальные конструкторы).
- Анализируя свойство ComponentState, конструктор может выявить флаг csDesigning, который помещается в это свойство, если компонент создается на стадии конструирования. Таким образом, в конструкторе можно разрешить или запретить некоторую функциональность компонента в зависимости от того, создается ли компонент в период прогона программы или на стадии конструирования формы. Например:

```
constructor TMyComp.Create(AOwner: TComponent);  
begin  
    inherited;  
    if csDesigning in ComponentState then  
        // Выполнение специфичных действий  
end;
```

- При перекрытии конструктора вначале вызывается унаследованный конструктор для распределения памяти и создания соответствующего объекта, только после этого выполняются специфические действия, например, установка начальных значений. При перекрытии деструктора наоборот — сначала выполняются действия по освобождению выделенных компоненту ресурсов, после чего вызывается унаследованный деструктор.

Создание событий

Создание событий имеет одну особенность, связанную с диспетчеризацией. Как и любое другое свойство, событие обычно имеет связанное с ним закрытое поле, в которое помещается адрес обработчика. Определив, тем или иным образом, факт наступления события, компонент должен проверить содержимое этого поля и, если в нем находится адрес обработчика, передать ему управление.

Определим для компонента TMyComp событие OnTimer, связанное со срабатыванием его внутреннего таймера (листинг 26.3).

Листинг 26.3. Диспетчеризация события OnTimer

```
TMyComp = class (TWinControl)  
    private  
        ...  
        FOnTimer: TNotifyEvent;  
    procedure OnInnerTimer(Sender: TObject);  
    public  
        constructor Create(AOwner: TComponent); override;  
        ...
```

```

    published
    ...
    property OnTimer: TNotifyEvent read FOnTimer write
FOnTimer;
    end;
    ...

    constructor TMyComp.Create(AOwner: TComponent);
    begin
    ...
    FTimer := TTimer.Create(Self);
    FTimer.OnTimer := OnInnerTimer;
    end;

    procedure TMyComp.OnInnerTimer(Sender: TObject);
    // Диспетчеризация события OnTimer
    begin
    if Assigned(FOnTimer) then
        FOnTimer(Sender)
    end;

```

Как и в процессе создания свойств, следует опубликовать необходимые события родительского класса. Например:

```

TMyComp = class(TWinControl)
    ...
    public
    ...
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
end;

```

Методика использования ModelMaker

Приводившиеся выше примеры наглядно демонстрируют, какой значительный объем кода сопровождает создание нового компонентного класса. ModelMaker во многом избавляет программиста от ручного кодирования свойств, методов и событий, сокращая сроки разработки.

В этом разделе на примере создания функционально законченного компонента иллюстрируется методика использования ModelMaker. Компонент, о котором пойдет речь, есть всего лишь специализированная метка TLabel, которая в своем свойстве Caption отображает строковое представление целого числа.

Этапы разработки

В ModelMaker используется понятие модели как аналог проекта в Delphi. Ниже перечислены основные этапы работы с ModelMaker.

1. Создается новая модель.
2. В модель помещается новый класс. При этом определяется родительский класс и указывается имя нового класса.
3. В классе определяются новые и/или перекрываются старые свойства, методы и события.
4. Для каждой новой сущности класса программист кодирует только операторы исполняемого кода.
5. Новый класс помещается в модуль.
6. Выполняется генерация вновь созданной модели, в результате которой получается полный исходный текст модуля. Модель и файл модуля сохраняются на диске.
7. С помощью Delphi осуществляется отладка нового модуля, и, если он содержит новые компоненты, эти компоненты регистрируются в VCL.
8. С помощью ModelMaker исходный текст документируется (снабжается комментариями). ModelMaker поможет также создать необходимый файл помощи.

В следующих подразделах каждый из указанных этапов рассматривается более подробно.

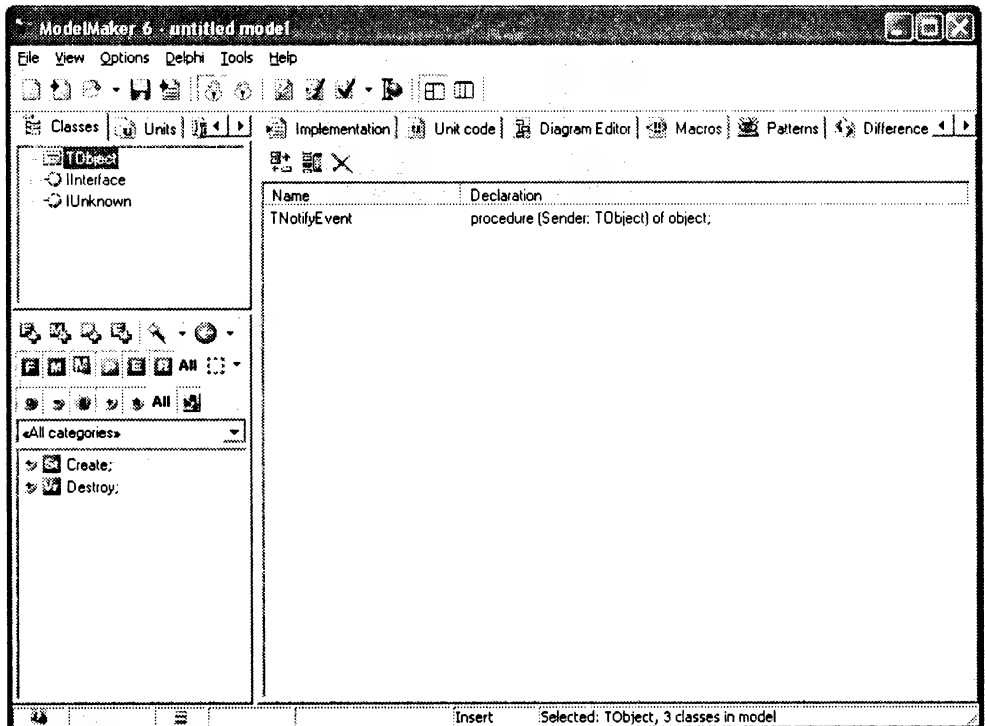


Рис. 26.2. Окно визуальной среды разработчика модели

Активизация ModelMaker

После установки ModelMaker в главное меню Delphi автоматически добавляется пункт ModelMaker. Выберите команду ModelMaker ► Run ModelMaker. На экране появится окно визуальной среды разработчика модели (рис. 26.2).

Верхняя часть окна содержит главное меню и инструментальные кнопки для манипуляции моделью как таковой.

Левая часть окна занята двумя браузером. Верхний управляется двумя вкладками, переводящими его в режим отображения дерева классов и интерфейсов (вкладка *Classes*) или связанных с моделью модулей (вкладка *Units*). Нижний отображает сущности выбранного класса (интерфейса).

Вся оставшаяся часть окна занята рабочей зоной, содержимое которой управляется вкладками в ее верхней части.

Вставка нового класса

Как уже отмечалось в предыдущем разделе, важным шагом при создании нового компонента является правильный выбор родительского класса. Для нашего примера таковым будет класс *TLabel*, для которого достаточно добавить новое свойство *NumValue*. Для правильной генерации нового класса в браузере классов следует указать сначала имя родительского класса. Это может показаться сложным: для родительского класса нужно указать его родителя, затем родителя родителя и т. д. К счастью, это не так. В ModelMaker для родительского класса нового класса можно указать любого родителя, например, *TObject* или *TComponent*.

1. Выберите команду *File ► New From Default*.
2. В окне браузера классов щелкните на классе *TComponent* и нажмите клавишу *Ins*.
3. Замените умалчиваемое имя *TDefault1* именем *TLabel*.
4. Вновь нажмите клавишу *Ins* для определения нового класса и замените его имя именем *TIntLabel*. После этого окно браузера должно выглядеть так, как показано на рис. 26.3.

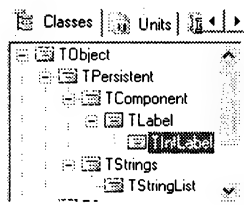


Рис. 26.3. Определение нового класса

Определение новых свойств и методов

В новом классе *TIntLabel* следует определить новое свойство и два метода для доступа к нему. Результирующий код должен быть приблизительно таким:

```
type
  TIntLabel = class (TLabel)
```

```

protected
  function GetNumValue: Integer;
  procedure SetNumValue(Value: Integer);
published
  property NumValue:
    Integer read GetNumValue write SetNumValue;
end;

```

Для добавления к классу новых членов используется нижний браузер, окно которого показано на рис. 26.4.

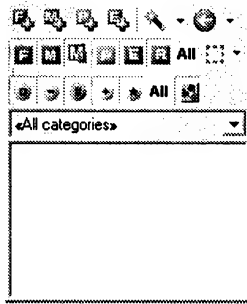


Рис. 26.4. Окно браузера членов класса

В его верхней части содержатся инструментальные кнопки для добавления новых полей, методов, свойств, событий и определения областей их видимости.

1. Щелкните на кнопке добавления свойства (третья слева в верхнем ряду). На экране появится окно определения свойства (рис. 26.5).
2. В поле Name введите имя свойства NumValue.
3. В группе Visibility установите переключатель published.
4. Убедитесь, что в группе Data type установлен переключатель Integer.
5. В группах Read Access и Write Access установите переключатели Methods и закройте окно щелчком на кнопке OK.

В окне браузера членов класса должны появиться названия двух методов (GetNumValue и SetNumValue) и свойства (NumValue).

Вставка исполняемого кода

Вы уже, очевидно, поняли, что значительную часть кода ModelMaker генерирует автоматически. Взгляните на конечный код двух методов:

```

function TIntLabel.GetNumValue: Integer;
begin
  Result := StrToIntDef(Caption, 0)
end;
procedure TIntLabel.SetNumValue(Value: Integer);
begin
  Caption := IntToStr(Value)
end;

```

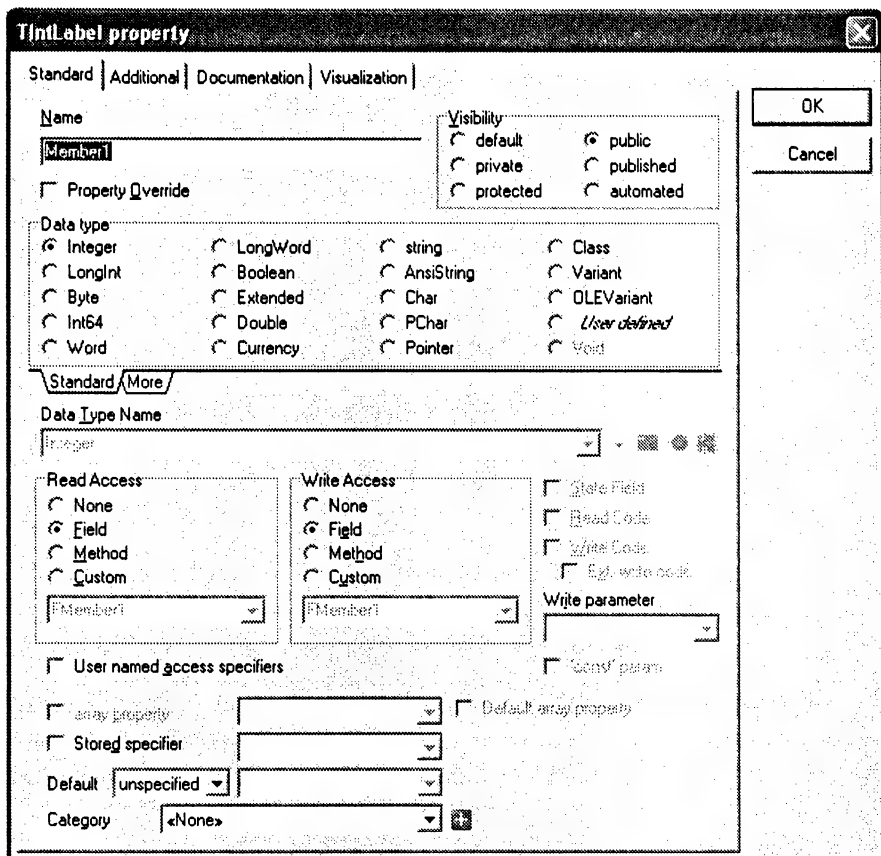


Рис. 26.5. Диалоговое окно определения нового свойства

Заголовки методов и зарезервированные слова **begin** и **end** вставляет ModelMaker. Программисту нужно вставить лишь две строки, определяющие исполняемую часть методов. Разумеется, во многих практически важных случаях вставляемые методы не будут такими простыми, как в нашем примере, и потребуют значительно большей ручной работы. Но и тогда экономия времени очевидна.

1. Щелкните на методе `GetNumValue` в окне браузера членов класса.
2. Перейдите на вкладку `Implementation` в рабочей зоне ModelMaker. Рабочая зона будет содержать набор инструментальных кнопок, окно браузера локального кода, окно результирующего кода и окно редактора кода (рис. 26.6). С помощью инструментальных кнопок в формируемый метод можно при необходимости вставить локальные переменные и локальные процедуры, которые будут тут же отображаться в окне браузера локального кода. Щелчок на локальной процедуре в этом браузере приведет к отображению локальной процедуры в окне редактора кода, что позволит сформировать ее исполняемую часть. Щелчок на окне результирующего кода активизирует текущую секцию этого кода в окне редактора кода.

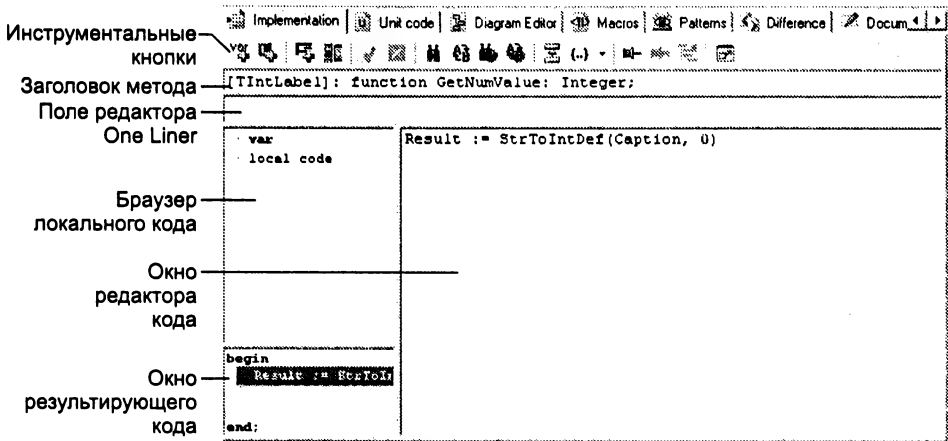


Рис. 26.6. Вид рабочей зоны в режиме ввода исполняемого кода

- Щелкните в окне редактора кода и введите следующую строку (отступы вводить не обязательно — их автоматически вставит ModelMaker):

```
Result := StrToIntDef(Caption, 0)ModelM
```

- Щелкните на кнопке **Save code** (пятая слева). В окне отображения результирующего кода появится только что введенный оператор.
- Щелкните на процедуре **SetNumValue** в окне браузера членов класса.
- В редакторе кода введите такой оператор:

```
Caption := IntToStr(Value)
```

Обратите внимание, что слева от операторов в окне результирующего кода появилась зеленая вертикальная линия. Это означает, что вами создана *секция кода*, которой вы можете распоряжаться по своему усмотрению. В код может вставляться неизменяемая секция, которая создана как образец, не подлежащий изменению. Такие секции помечаются красной линией. Наконец, количество отображаемых строк любой секции определяется настройками редактора — поле *Fold height* на вкладке *Editors* окна *Environment Options* (открывается командой *Options ► Environment Options*). Если в секции больше строк, она помечается второй линией пурпурного цвета.

Создание модуля с исходным кодом

Фактически, мы завершили формирование модели. Однако все внесенные в нее изменения остаются внутри ModelMaker. Чтобы показать их Delphi, необходимо создать модуль и сохранить этот модуль на диске. Результирующий код модуля показан в листинге 26.4.

Листинг 26.4. Код модуля *IntLabel.pas*

```
unit IntLabel;
interface
uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
```

```

type
  TIntLabel = class (TLabel)
  protected
    function GetNumValue: Integer;
    procedure SetNumValue(Value: Integer);
  published
    property NumValue:
      Integer read GetNumValue write SetNumValue;
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('MM Demo', [TIntLabel]);
end;
function TIntLabel.GetNumValue: Integer;
begin
  Result := StrToIntDef(Caption, 0);
end;
procedure TIntLabel.SetNumValue(Value: Integer);
begin
  Caption := IntToStr(Value);
end;
end.

```

1. Щелкните на вкладке Units окна браузера классов.
2. Щелкните на единственной доступной инструментальной кнопке Add. На экране появится окно редактора модуля (рис. 26.7).
3. Оставьте выбранным пункт <No alias> в списке Source path Alias.
4. Введите в поле Relative Unit file name полный маршрут доступа к файлу. Например, C:\MM Demo\IntLabel.PAS. Для формирования сложного маршрута используйте кнопку Browse.
5. В списке Available classes щелкните на классе TIntLabel и затем на кнопке с направленной вправо стрелкой.
6. В графе VCL Page списка Classes in unit введите имя MM Demo в качестве имени вкладки галереи компонентов Delphi, на которой должен разместиться создаваемый компонент (эта вкладка будет создана в момент регистрации компонента).
7. Щелкните на кнопке OK. В правой части рабочей зоны появится такой текст:

```

unit <!UnitName!>;
interface
uses
  SysUtils, Windows, Messages, Classes, Graphics,
  Controls, Forms, Dialogs;
type
  MMWIN: STARTINTERFACE

```

```

MMWIN:CLASSINTERFACE TIntLabel; ID=533;
procedure Register;
implementation
procedure Register;
begin
MMWIN:CLASSREGISTRATION TIntLabel; ID=533;
    Page='MM Demo';
end;
MMWIN:STARTIMPLEMENTATION
MMWIN:CLASSIMPLEMENTATION TIntLabel; ID=533;
end.

```

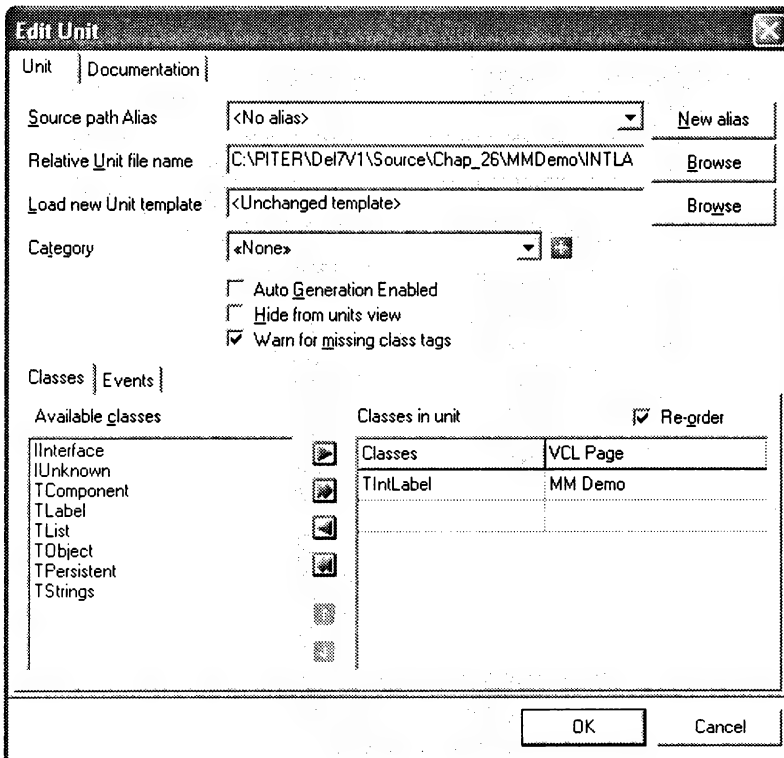


Рис. 26.7. Окно редактора модуля

Этот текст содержит различные *теги* (начинаются символами *MMWIN*), которые при генерации текста будут заменены стандартными операторами. В предложении **uses** перечислена стандартная последовательность модулей, которые обычно используются при создании нового модуля в Delphi. Родительский класс *TLabel* объявлен в модуле *StdCtrls*, которого в этой последовательности нет.

8. Добавьте ссылку на *StdCtrls* в предложение **uses** вручную.
9. Щелкните на кнопке **Save code**. Теперь все готово для генерации исходного модуля *IntLabel.pas*.

10. Щелкните на кнопке **Unlock code generation** (6-я слева в верхнем ряду).
11. В окне браузера модулей щелкните на кнопке **Generate** (3-я слева).

Теперь исходный код создан и находится на диске в указанной вами папке в файле с именем `IntLabel.pas`. Дальнейшая работа с модулем (отладка и регистрация компонента в VCL) должна происходить под управлением Delphi. Загрузите среду Delphi (если она еще не загружена) и нажмите клавиши `Ctrl+F11` или перейдите в Delphi вручную.

Отладка компонента

Особенностью отладки компонента является необходимость его регистрации в VCL. Однако если новый модуль имеет серьезные ошибки, это может привести к сбоям Delphi вплоть до полной потери работоспособности системы. Поэтому отладку компонента рекомендуется вести до его регистрации. Для этого создается новый проект, к которому присоединяется модуль с компонентом. В главном окне проекта размещаются все необходимые для проверки нового компонента элементы управления. Сам компонент не может быть создан на этапе конструирования программы (он еще не зарегистрирован в VCL), поэтому его следует создавать на этапе прогона программы (обычно в обработчике события `OnCreate` главной формы).

С помощью главного окна проекта `Source\Chap_26\MMDemo\TestMMDemo.dpr` тестируется компонент `TIntLabel`. Обратите внимание на такой фрагмент текста модуля главной формы:

```
var
    IntLabel: TIntLabel;
procedure TForm1.FormCreate(Sender: TObject);
begin
    IntLabel := TIntLabel.Create(Self);
    IntLabel.Parent := Self;
end;
```

В обработчике события `OnCreate` формы не только создается объект класса `TIntLabel`, но и в его свойство `Parent` (Родитель) помещается ссылка на объект формы. Только в этом случае можно синхронизировать прорисовку главной формы и всех ее дочерних компонентов (см. раздел «Родительские и дочерние компоненты» в главе 14).

Совершенствование компонента

В результате тестирования компонента могут выявиться те или иные недочеты, требующие исправления. Для исправления компонента можно, разумеется, внести исправления в модуль `IntLabel` средствами редактора кода Delphi, но естественнее использовать возможности ModelMaker. В этом последнем случае исходный код будет сгенерирован заново, и все внесенные в него редактором Delphi изменения *окажутся потерянными*.

Например, наш компонент `TIntLabel` не виден в начале прогона программы вплоть до того, как мы не пошлем ему число или текст. Для исправления ситуации следует перекрыть его конструктор, чтобы в его свойство `NumValue` поместить 0.

Более того, если бы мы зарегистрировали компонент в его теперешнем виде, он стал бы доступен на этапе конструирования, и при помещении его на форму среда Delphi поместила бы в его свойство `Caption` собственное имя компонента (вроде `IntLabel1`). Таким образом, в перекрытом конструкторе мы должны запретить изменение его свойства `Caption` на этапе конструирования программы. В результате конструктор должен иметь такой вид (см. раздел «Свойства разного назначения» в главе 14):

```
constructor TIntLabel.Create(AOwner: TComponent);  
begin  
    inherited Create(AOwner);  
    // Запрещаем изменение свойства Caption  
    // на этапе конструирования  
    ControlStyle := ControlStyle - [csSetCaption];  
    // Устанавливаем свойство NumValue  
    NumValue := 0;  
end;
```

Вернемся к ModelMaker.

1. Щелкните на кнопке **Wizards** в окне браузера членов класса (5-я в верхнем ряду).
2. В новом окне (рис. 26.8) щелкните на строке `Create(Owner: TComponent)` и убедитесь, что установлен флажок **Call inherited method**.

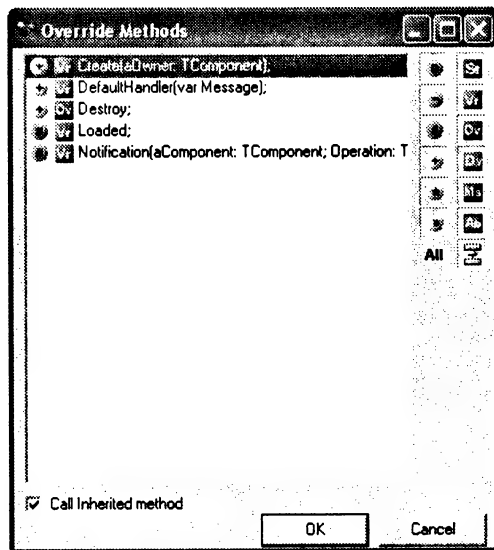


Рис. 26.8. Окно перекрытия метода

3. Закройте окно щелчком на кнопке **OK**.
4. Перейдите на вкладку **Implementation** в рабочей зоне. Вы увидите, что средой ModelMaker была вставлена неизменяемая секция кода с единственным оператором:


```
inherited Create(AOwner);
```

5. Добавьте новую секцию кода щелчком на кнопке Add section.
6. Введите в редакторе кода следующий текст и сохраните его щелчком на кнопке Save code:

```
// Запрещаем изменение свойства Caption на этапе
// конструирования
ControlStyle := ControlStyle - [csSetCaption];
// Устанавливаем свойство NumValue
NumValue := 0;
```

Для регенерации исходного кода можно было бы еще раз щелкнуть на кнопке Generation, но у ModelMaker есть более гибкая возможность: щелкните на кнопке Enable Auto Generation в окне браузера модулей и затем вызовите Delphi. Вы заметите, что файл IntLabel.pas, представленный в окне кодового редактора, автоматически заменен новым. Пример тесного взаимодействия ModelMaker с Delphi!

Оформление компонента

В этом небольшом подразделе рассказывается о рутинной, но весьма нужной работе над вновь созданным компонентом. Конкретно мы поговорим:

- о документировании исходного листинга средствами ModelMaker;
- о создании значка компонента, который будет виден на нужной вкладке палитры компонентов;
- о создании справочной службы компонента и ее интеграции со справочной службой Delphi.

Документирование исходного листинга

ModelMaker позволяет в определенной мере автоматизировать и унифицировать процесс создания документированного листинга нового компонента. К сожалению, автоматически создаваемые этим инструментом строки рассчитаны на англоязычного пользователя.

Чтобы продемонстрировать возможности ModelMaker, перейдите в эту среду и щелкните на вкладке Documentation в рабочей зоне. Затем щелкните на кнопке Wizard и согласитесь с предложением мастера произвести стандартную генерацию документации. Щелкайте поочередно на каждом члене класса TIntLabel в окне браузера и вы будете видеть представленный ниже текст.

- Для метода GetNumValue (GetNumValue — метод чтения свойства NumValue):
GetNumValue is the read access method of the NumValue property
- Для метода SetNumValue (SetNumValue — метод записи в свойство NumValue):
SetNumValue is the write access method of the NumValue property
- Для свойства NumValue (свойство NumValue доступно для чтения/записи на этапе прогона и конструирования):

Property NumValue is read / write at run time and design time

- ☛ Для конструктора Create (конструктор Create перекрывает унаследованный конструктор Create; вначале вызывается унаследованный конструктор, затем инициализируется внутренняя структура данных):

Constructor Create overrides the inherited Create.
First inherited Create is called, then the internal data structure is initialized

Согласитесь, это не так уж плохо. Но для русскоязычного пользователя такая документация может не подойти. В этом случае вы можете в окне редактора на вкладке Documentation вставить любой текст (см. проект Source\Chap_26\MMDemo\Project1.mpb).

Создание значка

Для создания значка воспользуйтесь редактором ImageEditor, входящим в поставку Delphi (выбрав команду Tools ► Image Editor). Создайте с его помощью растровое изображение (команда File ► New ► Component Resource File и затем Resource ► New ► Bitmap) размером 24 × 24 в цветовой палитре из 16 цветов. Назовите новый ресурс именем класса компонента (TINTLABEL), причем *обязательно используйте прописные буквы*, после чего сохраните ресурсный файл в той же папке, где расположен модуль компонента, назовите его именем модуля с расширением DCR (IntLabel.dcr). После этого в момент регистрации ресурс будет автоматически связан с компонентом.

Создание справочной службы

ModelMaker автоматизирует процесс создания справочной службы для разработанного компонента. Для этого ModelMaker использует созданную этой средой и/или программистом документацию и представляет ее в виде готового к компиляции RTF-файла и HPL-файла проекта. ModelMaker автоматически включает в заголовки тем (разделов) справки ссылки В, обеспечивающие необходимую интеграцию HPL-файла со справочной службой среды Delphi.

Для генерации файлов, необходимых для создания справочной службы, в раскрывающемся списке на вкладке Documentation выберите пункт Units и щелкните на небольшой кнопке Help file слева от списка (эта кнопка недоступна при любом другом выбранном в списке пункте). В появившемся диалоговом окне (рис. 26.9) вы должны указать папку для размещения файлов и имя RTF-файла.

С помощью переключателей группы Class members Visibility вы можете определить области видимости членов класса, информация о которых будет помещена в справочную службу:

- ☛ User (public, published, automated) — информация рассчитана на пользователя; документируются члены, располагающиеся в секциях **public**, **published**, **automated**;
- ☛ Component writer (User plus protected) — информация рассчитана на программистов, которые будут создавать наследников вашего компонента; документация дополняется членами, объявленными в секции **protected**;

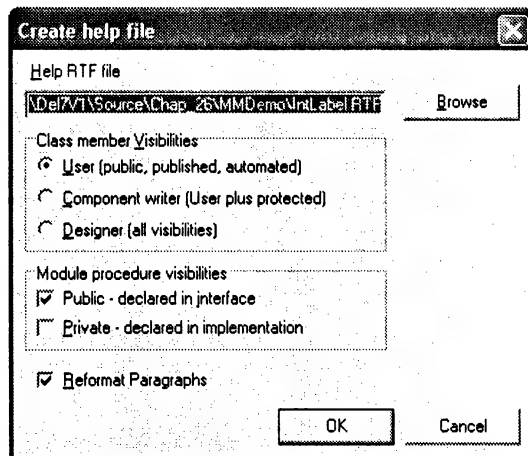


Рис. 26.9. Окно настройки параметров генерации справочной службы

- **Designer (all visibilities)** — документируются все члены класса.

С помощью флажков группы **Module procedure visibilities** выбираются процедуры модуля, которые будут документированы:

- **Public - declared in interface** — процедуры, объявленные в интерфейсной части модуля;
- **Private - declared in implementation** — процедуры, объявленные в исполняемой части.

Флажок **Reformat Paragraphs** обычно установлен и обязывает ModelMaker преобразовать в единый абзац все строки документации. Если он сброшен, строки не преобразуются.

После задания необходимых параметров щелкните на кнопке **OK**.

СОВЕТ При преобразовании строк ModelMaker использует кодировку КОИ-7, согласно которой старший бит каждого символа отбрасывается. Поскольку символы кириллицы занимают коды от 128 и выше, русскоязычная документация будет безнадежно испорчена. В этом случае можно порекомендовать ручной (через буфер обмена) перенос текстовых фрагментов с вкладки **Documentation** в результирующий RTF-файл.

Для генерации HLP-файла используется компилятор **HCW.EXE**, который располагается в папке **Help\Tools** каталога размещения Delphi. С помощью команды **File ► Open** или щелчком на соответствующей инструментальной кнопке откройте созданный ModelMaker файл проекта **IntLabel.hpj**, после чего щелчком на инструментальной кнопке **Compile** получите файл справки.

Регистрация компонента

После отладки компонента и создания его значка можно зарегистрировать компонент в палитре компонентов Delphi. Для этого в среде Delphi выберите команду **Component ► Install Component**. По умолчанию Delphi предлагает перенести компо-

нент в пакет¹ Lib\dcluser.dpk каталога размещения Delphi. Если вы хотите установить компонент в собственный пакет, перейдите на вкладку Into new package (рис. 26.10).

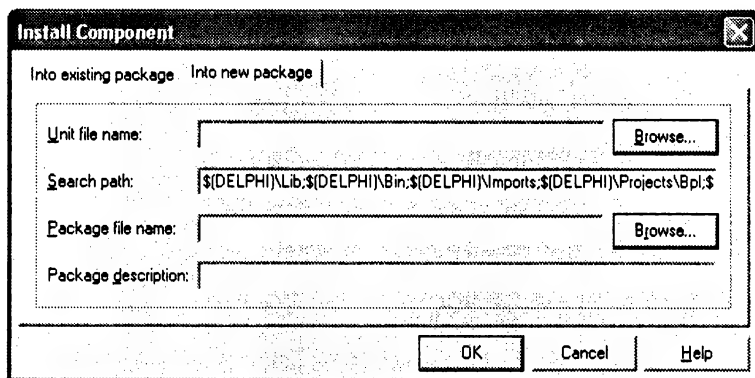


Рис. 26.10. Регистрация компонента в новом пакете

В поле Unit file name следует указать имя файла с модулем компонента. Советую использовать кнопку Browse, так как в этом случае не только упрощается указание маршрута доступа к файлу, но и этот маршрут автоматически дополняет список Search path, в котором перечисляются папки для поиска связанных с компонентом ресурсов. В поле Package file name нужно указать имя вновь создаваемого файла пакета. Поле Package description может содержать произвольную текстовую строку, которая будет идентифицировать пакет в списке всех зарегистрированных пакетов.

После заполнения полей и щелчка на кнопке OK осуществляется компиляция модуля (если он не был предварительно откомпилирован). В ходе регистрации автоматически ищется файл с компонентным ресурсом (DCR). Если он не был найден (а это может быть, если вы не назвали его по имени компонентного модуля или компонентного класса), компоненту будет присвоен значок родительского класса или специальный значок, если родительский класс не имеет значка. После успешной регистрации вы можете увидеть компонент на вкладке, определенной первым параметром обращения к процедуре RegisterComponents в модуле компонента (если до регистрации компонента такой вкладки не существовало, она будет создана).

При экспериментах с компонентами вам может понадобиться отказаться от регистрации компонента. Для этого необходимо в контекстном меню палитры компонентов выбрать команду Properties, в поле Pages открывшегося окна Palette Properties активизировать нужную страницу и щелкнуть на кнопке Delete.

Для связывания файла справки компонента со справочной системой Delphi выберите команду Help ► Customize, в открывшемся окне перейдите на вкладку Link и щелкните на кнопке Add Files или выберите команду Edit ► Add Files.

¹ Пакетами в Delphi называются особым образом оформленные библиотеки DLL, предназначенные для хранения компонентов.

Приложение А

Интегрированная среда разработчика

Интегрированная среда разработчика (ИСР) Delphi содержит огромное количество вспомогательных утилит и диалоговых окон, описать которые в одном приложении просто невозможно — для этого требуется отдельная книга. В этом приложении мы познакомимся с командами главного меню, а также особенностями работы с текстовым редактором и возможностями отладки.

Главное меню

В этом разделе описываются главное меню Delphi 7, а также наиболее важные инструменты этой среды (включая хранилище объектов, менеджер проекта, менеджер трансляций и т. д.) и приемы работы с ней (включая управление группой компонентов, выбор языков локализации, настройка параметров среды и пр.). Следует сказать, что некоторые пункты главного меню появляются только после установки необязательных утилит. Эти (необязательные) пункты не рассматриваются.

Меню File

Команды меню File перечислены ниже.

- **New** — подменю со следующими командами:
 - ☐ **Application** — создает новую программу для Windows;
 - ☐ **CLX Application** — создает новую программу для Windows и Linux;
 - ☐ **Data Module** — создает новый модуль данных;
 - ☐ **Form** — создает новую форму и подключает ее к проекту;
 - ☐ **Frame** — создает новую рамку;
 - ☐ **Other** — открывает окно хранилища объектов (см. ниже).
- **Open** — открывает ранее созданную форму.
- **Open Project** — открывает ранее созданный проект.

- Reopen — вызывает список ранее загружавшихся проектов и форм для выбора и повторной загрузки.
- Save — сохраняет активную форму.
- Save As — сохраняет активную форму под другим именем.
- Save Project As — сохраняет файл проекта под другим именем.
- Save All — сохраняет файл проекта и все открытые модули.
- Close — закрывает текущую форму.
- Close All — закрывает все открытые файлы.
- Use Unit — вставляет в текущую форму ссылку на другой модуль.
- Print — печатает активную форму или модуль.
- Exit — прекращает работу Delphi.

Хранилище объектов

Окно хранилища объектов Delphi (рис. А.1) открывается при выборе команды File ► New ► Other.

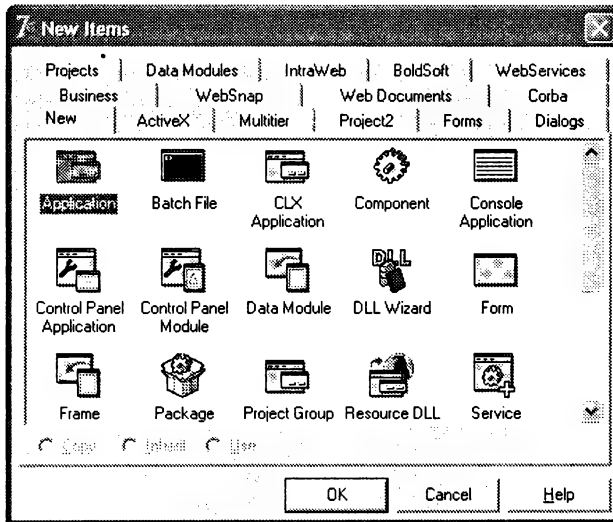


Рис. А.1. Окно хранилища объектов Delphi

На вкладках этого окна расположены значки для выбора прототипов форм, модулей, проектов и мастеров (экспертов) построения форм и проектов. Переключатели Copy, Inherit и Use определяют режим связи между хранящимся в хранилище прототипом и его копией в проекте:

- Copy — выбранный элемент копируется в текущий каталог и автоматически подключается к проекту; между прототипом и его копией нет никакой связи;
- Inherit — в проекте создаются наследники выбранного элемента и всех его родителей; любые изменения прототипа проявляются во всех проектах, которые его унаследовали; изменения наследника не влияют на прототип;

- Use — выбранный элемент становится частью проекта; любые его изменения в проекте приводят к изменениям прототипа и сказываются во всех других проектах, которые его унаследовали или используют.

Ниже кратко описывается методика размещения в хранилище разработанной программистом формы.

1. Разработайте форму, которую вы часто будете использовать в *разных* проектах¹. Поместите на ней все необходимые интерфейсные компоненты и, если это необходимо, закодируйте обработчики их событий.
2. Сохраните форму в любой папке жесткого диска. Для упорядочения поиска прототипов можно использовать для этих целей папку Objrepos каталога размещения Delphi.
3. Щелкните на форме правой кнопкой мыши и в контекстном меню выберите команду Add to Repository. На экране появится диалоговое окно регистрации формы в хранилище (рис. А.2).
4. В поле Title введите имя, под которым форма будет зарегистрирована в хранилище, например, WorkForm, в поле Description — описание формы, например, Основная форма для проекта Bibliophil, в поле Author — свое имя.
5. Раскройте список Page и выберите ту вкладку хранилища, на которой следует разместить форму, после чего щелкните на кнопке OK — форма окажется в хранилище.

ВНИМАНИЕ Если вы не укажете страницу, форма не будет размещена в хранилище, однако вы сможете ее найти на вкладке с именем вашего проекта. Тиражирование такой формы, как и любой другой формы этой страницы, возможно только в рамках вашего проекта.

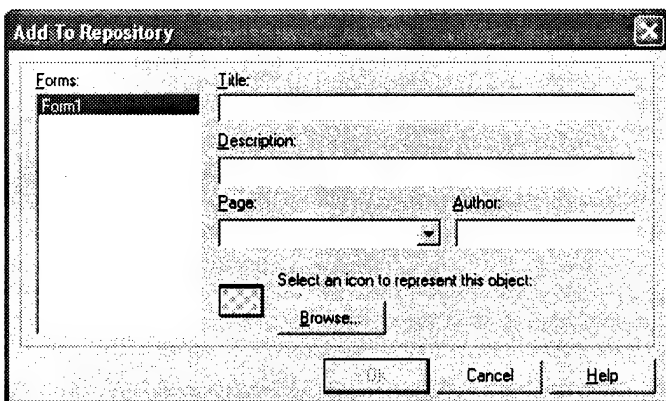


Рис. А.2. Окно регистрации формы в хранилище объектов

¹ Если вы хотите тиражировать какую-то форму текущего проекта, ее нет нужды размещать в хранилище — она там уже есть на вкладке с именем вашего проекта (такая вкладка в окне хранилища автоматически создается для каждого открываемого проекта).

Вновь размещенную форму можно сделать главной. Главная форма создается при старте нового проекта с помощью команды **File ▶ New ▶ Application**. Для определения главной формы выберите команду **Tools ▶ Repository**, в открывшемся диалоговом окне укажите вкладку размещения в списке **Pages** и форму в списке **Objects**, после чего установите флажок **Main Form**. Если установить флажок **New Form**, выбранная форма будет создаваться при щелчке на кнопке **New Form** панели инструментов или при выборе команды **File ▶ New ▶ Form**.

Меню Edit

Команды меню **Edit** перечислены ниже.

- ✎ **Undo** — отменяет последнее изменение проекта.
- ✎ **Redo** — восстанавливает последнее изменение проекта.
- ✎ **Cut** — вырезает выбранный компонент формы или фрагмент текста и помещает его в буфер обмена (clipboard).
- ✎ **Copy** — копирует в буфер обмена выделенные компоненты формы или фрагмент текста модуля.
- ✎ **Paste** — извлекает из буфера обмена и переносит компоненты на форму или копирует текст в модуль (в позицию, идентифицируемую текущим положением текстового курсора).
- ✎ **Delete** — удаляет выделенные компоненты или фрагмент текста.
- ✎ **Select All** — выделяет все компоненты формы или весь текст модуля.
- ✎ **Align To Grid** — привязывает выделенные компоненты к масштабной сетке так, чтобы их левые верхние углы располагались в ближайших точках сетки.
- ✎ **Bring To Front** — перемещает выделенные компоненты на передний план.
- ✎ **Send To Back** — перемещает выделенные компоненты на задний план.
- ✎ **Align** — вызывает окно выравнивания выделенных компонентов (см. ниже).
- ✎ **Size** — вызывает окно изменения размеров выделенных компонентов (см. ниже).
- ✎ **Scale** — масштабирует выделенные компоненты (см. ниже).
- ✎ **Tab Order** — изменяет порядок обхода компонентов, выполняющийся по нажатию клавиши **Tab**.
- ✎ **Creation Order** — изменяет порядок создания невидимых компонентов.
- ✎ **Flip Children** — изменяет свойство **BiDiMode** для всех или только для выделенных компонентов.
- ✎ **Lock Controls** — блокирует возможность перемещения компонентов на форме.
- ✎ **Add To Interface** — определяет новые свойства, методы и события для компонентов **ActiveX**.

Управление группой компонентов

Показанные на рис. А.3–А.5 окна предназначены для обработки группы выделенных на форме компонентов по образцу (или относительно) первого выделенного.

Чтобы выделить группу компонентов, нужно нажать и удерживать клавишу Shift, после чего щелкнуть на компонентах левой кнопкой мыши. Каждый отмеченный таким образом компонент выделяется маркерами (серыми квадратиками) по углам его видимых границ. Затем вызывается команда Edit ► Align, Edit ► Size или Edit ► Scale соответственно для выравнивания, изменения размеров или масштабирования компонентов. Другой вариант — не отпуская клавиши Shift, щелкнуть правой кнопкой мыши на любом свободном месте формы и выбрать нужную команду в контекстном меню.

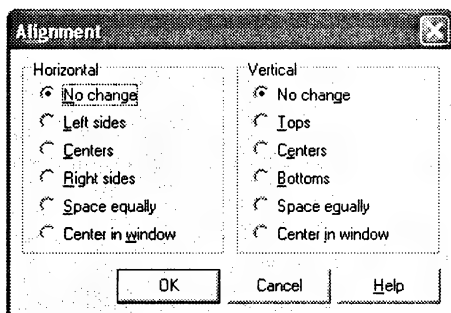


Рис. А.3. Окно выравнивания компонентов

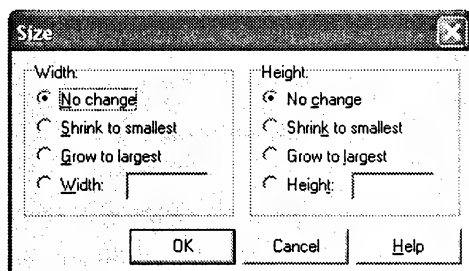


Рис. А.4. Окно изменения размеров компонентов

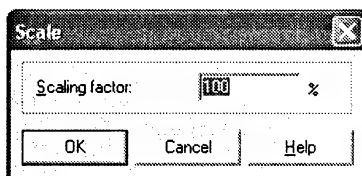


Рис. А.5. Окно масштабирования компонентов

Переключатели в окне выравнивания компонентов позволяют выбрать вариант выравнивания всех выделенных компонентов относительно самого первого выделенного компонента (эталона):

- No change — компоненты не выравниваются;
- Left sides — компоненты выравниваются по левой границе;
- Centers — компоненты центрируются относительно границ эталона;

- Right sides — компоненты выравниваются по правой границе эталона;
- Space equally — между всеми компонентами устанавливается равное расстояние по горизонтали или вертикали;
- Center in window — все компоненты центрируются относительно границ окна;
- Tops — компоненты выравниваются по верхнему краю;
- Bottoms — компоненты выравниваются по нижнему краю.

Группа переключателей Width в окне изменения размеров компонентов регулирует ширину выделенных компонентов, а группа переключателей Height — их высоту:

- No change — размеры компонентов не меняются;
- Shrink to smallest — уменьшает размеры компонентов до размеров наименьшего из них;
- Grow to largest — увеличивает размеры компонентов до размеров наибольшего из них;
- Width — позволяет задать ширину компонентов;
- Height — позволяет задать высоту компонентов.

В поле Scaling factor окна Scale вводится коэффициент масштабирования компонентов в процентах от текущих размеров.

Меню Search

Команды меню Search перечислены ниже.

- Find — ищет фрагмент текста и подсвечивает его, если он найден.
- Find In Files — ищет фрагмент текста во всех файлах проекта, только в открытых файлах или, наконец, во всех файлах текущего каталога.
- Replace — ищет и заменяет фрагмент текста.
- Search Again — повторяет поиск или поиск и замену.
- Incremental Search — ищет текст по мере его ввода — сначала первую букву, затем две первых буквы и т. д.
- Go to Line Number — перемещает курсор на строку с указанным номером от начала файла.
- Find Error — по адресу ошибки периода прогона программы отыскивает фрагмент кода, связанный с ее возникновением.
- Browse Symbol — показывает место определения символа программы (команда доступна только после успешного прогона программы). Символом считается любой глобальный идентификатор вашего проекта.

Меню View

Команды меню View перечислены ниже.

- Project Manager — показывает окно менеджера проекта.
- Translation Manager — открывает доступ к менеджеру трансляций.
- Object Inspector — показывает окно инспектора объектов.

- Object TreeView — показывает окно дерева объектов.
- To-Do List — открывает доступ к списку заданий.
- Alignment Palette — показывает палитру выравнивания компонентов.
- Browser — показывает окно браузера объектов.
- Code Explorer — показывает скрытое окно браузера кода.
- Component List — показывает окно выбора компонентов.
- Window List — подменю со списком открытых окон проекта.
- Additional Message Info — вызывает диалоговое окно для подготовки файла замечаний, чтобы переслать его на сайт Borland.
- Debug Windows — подменю отладочных окон со следующими командами:
 - ☐ Breakpoints — показывает окно точек останова;
 - ☐ Call Stack — показывает окно стека;
 - ☐ Watching Expressions — показывает окно наблюдения за переменными/выражениями;
 - ☐ Local Variables — позволяет наблюдать за изменениями локальных переменных в отладочном режиме;
 - ☐ Threads — показывает окно статуса потоков команд;
 - ☐ Modules — показывает окно модулей проекта;
 - ☐ Event Log — показывает журнал событий;
 - ☐ CPU — показывает состояние регистров центрального процессора;
 - ☐ FPU — показывает состояние регистров арифметического сопроцессора.
- Desktops — подменю управления конфигурациями основных окон со следующими командами:
 - ☐ Save Desktop — сохраняет текущую конфигурацию;
 - ☐ Delete — удаляет ранее сохраненную конфигурацию;
 - ☐ Save Debug Desktop — определяет текущую конфигурацию как отладочную.
- Toggle Form или Toggle Unit — переключает активность из окна формы в окно кода программы и обратно.
- Units — показывает окно модулей.
- Forms — показывает окно форм.
- Type Library — показывает окно библиотеки типов.
- New Edit Window — открывает новое окно с кодом текущего модуля.
- Toolbars — показывает окно настроек панелей инструментов.

Отладочные окна подробнее рассматриваются в разделе «Отладка программ». Некоторые другие окна, вызываемые командами меню View, описываются ниже.

Менеджер проекта

Центральную часть окна менеджера проекта (рис. А.6) занимает список всех связанных с проектом форм. Кнопка New открывает доступ к хранилищу, позволяя добавить новый модуль к проекту. Кнопка Remove удаляет модуль из проекта. После

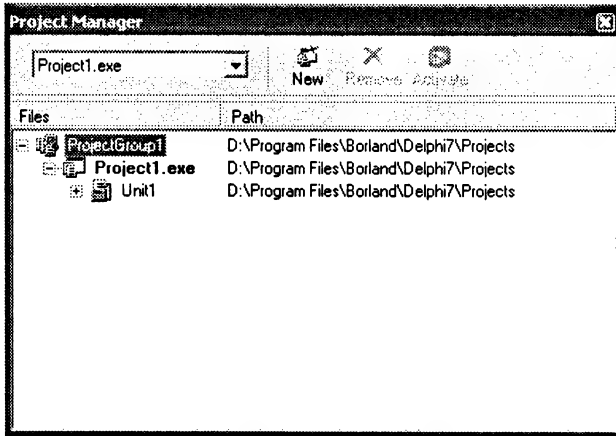


Рис. А.6. Окно менеджера проекта

щелчка правой кнопкой мыши на имени модуля появляется контекстное меню, с помощью которого можно открыть модуль или сохранить его под другим именем.

С помощью менеджера проектов осуществляется компиляция проекта на тот или иной язык локализации (см. ниже пункт «Менеджер трансляций»). Если для проекта определены языки локализации, в группе проектов помимо исполняемого файла Имя_проекта.EXE будут дополнительные проекты с тем же именем, но разными расширениями (по одному проекту на каждый язык локализации). Если, например, язык локализации английский (Великобритания), в группе будет проект Имя_проекта.ENG. Сделайте активным нужный проект и выберите команду Project ► Build Имя_проекта. В результате будет откомпилирован нужный ресурсный файл. После этого вновь активизируйте проект Имя_проекта.EXE, с помощью команды Project ► Language ► Set Active укажите язык локализации и вновь откомпилируйте проект. К проекту будет подключен нужный ресурсный файл, и ваша программа станет локализованной.

Менеджер трансляций

Менеджер трансляций упрощает создание локализованных версий программных продуктов. Он становится доступным только после указания языков, на которые будут переводиться текстовые сообщения, надписи, имена интерфейсных элементов и другие текстовые ресурсы программы. Для каждого языка создается своя динамически подключаемая библиотека ресурсов. Изменение этой библиотеки перед компиляцией программы изменяет ее язык (см. выше подраздел «Менеджер проекта»).

Для выбора языка (языков) локализации используется команда Project ► Language (см. пункт «Выбор языков локализации» в подразделе «Меню Project»). Главным языком программы по умолчанию считается язык локализации Windows, так что если Delphi работает под управлением русскоязычной версии Windows, главным языком будет русский.

Окно менеджера трансляций показано на рис. А.7. Для представленного на нем примера главным языком программы является русский, а языком локализации — английский (Великобритания).

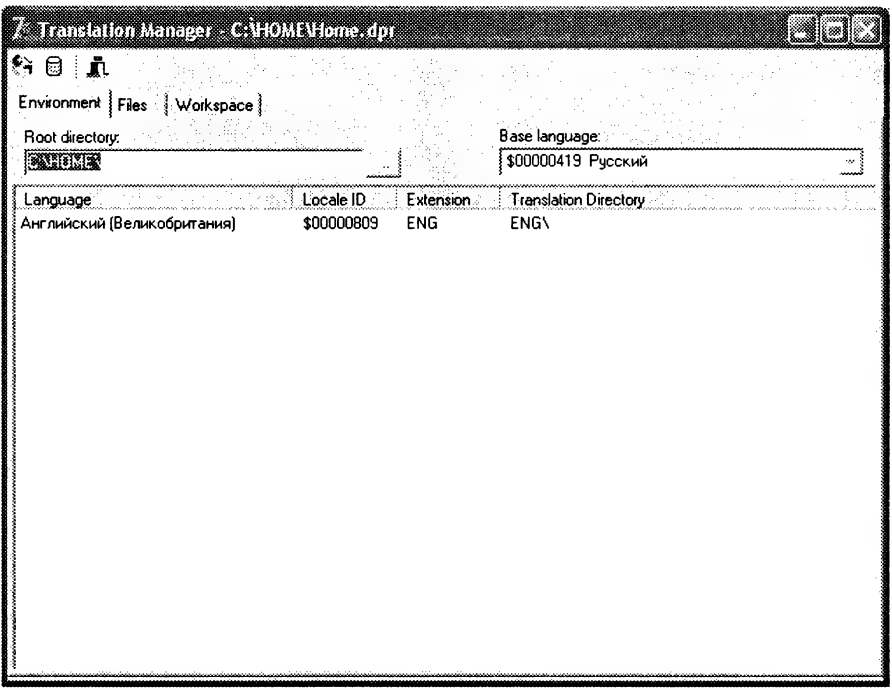


Рис. А.7. Окно менеджера трансляций

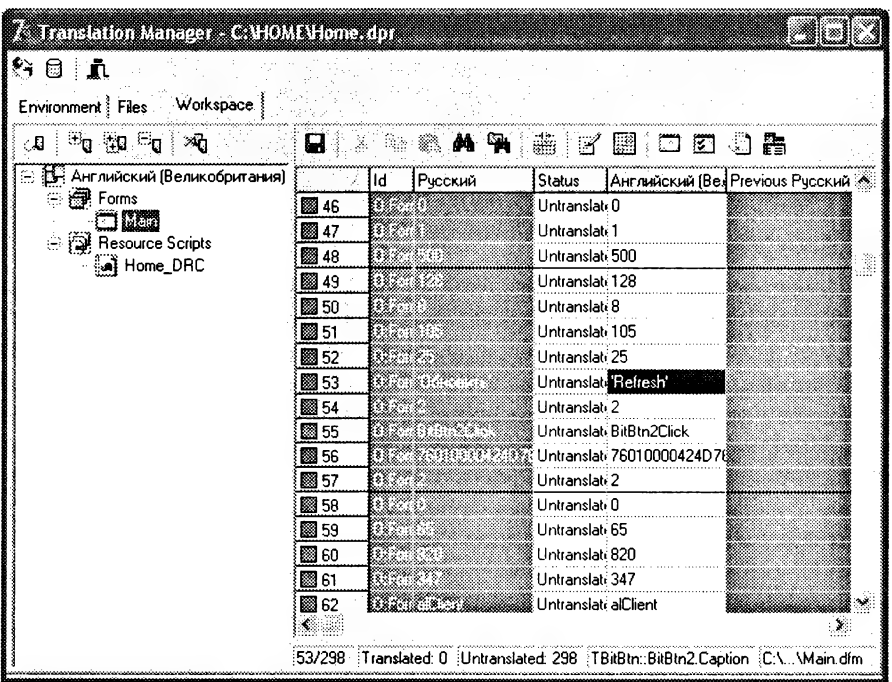


Рис. А.8. Вкладка Workspace менеджера трансляций

В окне менеджера трансляций имеются три вкладки. На вкладке **Environment** содержится список всех языков локализации (их может быть несколько). Вкладка **Files** отображает все ресурсы проекта, которые следует локализовать. Обычно это ресурсные файлы (DRC) и файлы описания форм (DFM). Вкладка **Workspace** (рис. А.8) содержит две панели.

Левая панель отображает древовидную структуру проекта. Перемещаясь по этому дереву (раскрывая соответствующие узлы), можно выбрать нужный ресурс. Его содержимое тут же отображается в правой панели. Теперь можно осуществить трансляцию строк (на рис. А.8 переведен термин **Обновить**).

Список заданий

Список заданий предназначен для координации работы нескольких программистов в рамках одного проекта. Этот список отображает все комментарии проекта, которые начинаются символами `//TODO` (рис. А.9). Комментарии содержат сообщения руководителя проекта и/или программистов об обнаруженных ошибках и обычно располагаются там, где обнаружена неточность. После вызова списка заданий переход к нужному комментарию осуществляется двойным щелчком мыши на соответствующей строке списка. Программист может сообщить об устранении ошибки, установив флажок в левой части строки списка — в этом случае текст строки выводится перечеркнутым шрифтом.

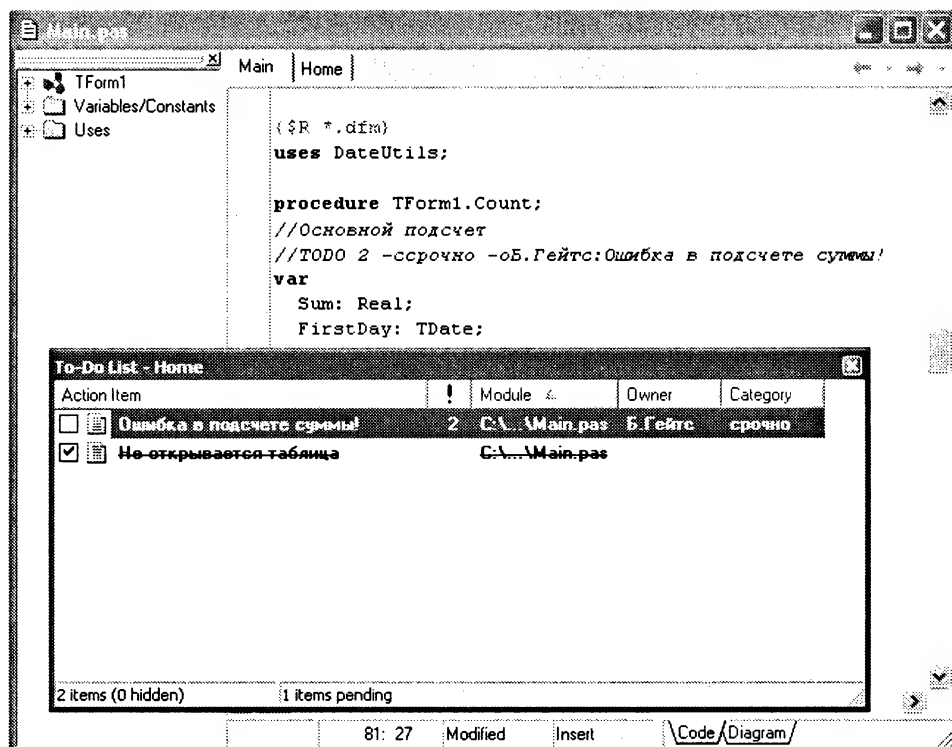


Рис. А.9. Пример списка заданий

В списке заданий можно указать приоритет сообщения (колонка «!»), его автора (Owner) и категорию (Category). Эти параметры можно задать с помощью соответствующих ключей непосредственно в комментарии или в окне редактора сообщения (оно вызывается после активизации сообщения и нажатия клавиши F2).

Браузер объектов

Браузер объектов (рис. А.10) доступен только после успешного прогона программы. Он представляет в наглядной форме используемые в проекте и доступные объекты, позволяя просмотреть иерархию объектов, а также их свойства и методы.

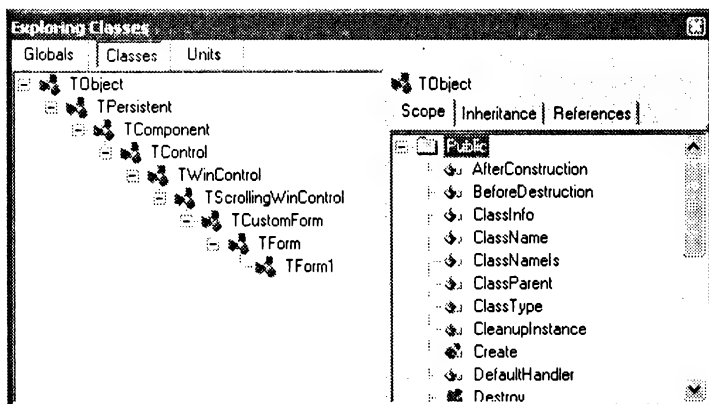


Рис. А.10. Окно браузера объектов

Меню Project

Команды меню Project перечислены ниже.

- Add To Project — добавляет файл к проекту.
- Remove From Project — удаляет файл из проекта.
- Import Type Library — импортирует в проект библиотеку типов элементов ActiveX.
- Add To Repository — помещает проект в хранилище.
- View Source — показывает окно с кодом проекта.
- Languages — подменю, которое позволяет добавить новый, удалить или сделать главным один из существующих языков локализации. Команды в этом подменю следующие:
 - ☐ Add — добавляет новый язык локализации (см. ниже);
 - ☐ Remove — удаляет существующий язык локализации;
 - ☐ Set Active — делает активным язык локализации;
 - ☐ Update Resources DLL — создает заново ресурсные библиотеки DLL, управляющие локализацией программы.
- Add New Project — добавляет программу, DLL или пакет к текущей проектной группе.

- Add Exists Project — открывает проект и добавляет его к текущей проектной группе.
- Compile Имя_проекта — компилирует модули, которые изменились с момента предыдущей компиляции проекта.
- Build Имя_проекта — компилирует все модули проекта и создает исполняемую программу.
- Syntax Check Имя_проекта — проверяет синтаксическую правильность программы.
- Information — показывает информацию о вашей программе.
- Compile All Projects — компилирует все файлы данной проектной группы, которые изменились с момента предыдущей компиляции.
- Build All Projects — компилирует все файлы данной проектной группы независимо от того, изменялись они с момента последней компиляции или нет.
- Web Deployment Options — устанавливает компонент ActiveX или ActiveForm на вашем веб-сервере. Вызывается перед компиляцией проекта.
- Web Deploy — устанавливает компонент ActiveX или ActiveForm на вашем веб-сервере. Вызывается после компиляции проекта.
- Options — показывает диалоговое окно установки параметров проекта.

Выбор языков локализации

После выбора команды Project ► Languages ► Add открывается окно эксперта по выбору языков локализации проекта (рис. А.11).

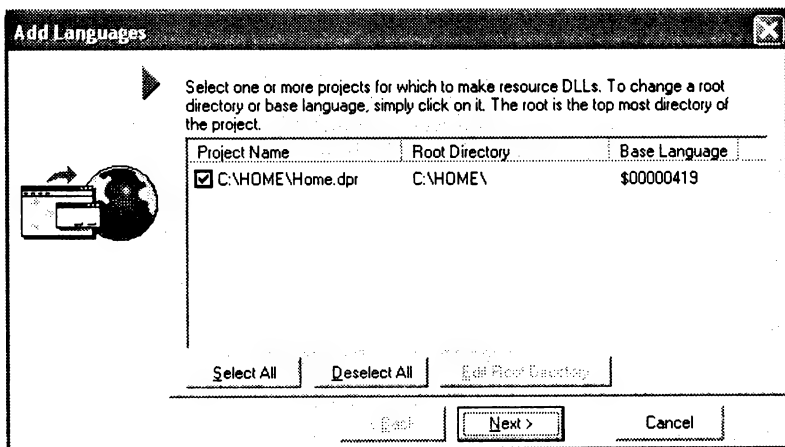


Рис. А.11. Окно выбора языков локализации

В окне указывается один или несколько проектов, для которых осуществляется локализация, и главный язык проекта (код \$000000419 соответствует русскому языку). В следующем окне (после щелчка на кнопке Next) вы сможете выбрать языки локализации, отметив их флажками. На следующем шаге эксперт предложит вам указать каталоги размещения библиотек, которые будут содержать локализованные ресурсы. Все библиотеки имеют одинаковое название Имя_проекта.DRC,

поэтому они должны размещаться в разных каталогах. Для компиляции программы с тем или иным языком локализации используется менеджер проекта (см. пункт «Менеджер проекта» в подразделе «Меню View») и команда Project ► Language ► Set Active.

Управление параметрами проекта

Управление параметрами проекта осуществляется с помощью диалогового окна, вызываемого командой Project ► Options.

На вкладке Forms окна параметров проекта (рис. А.12) выбираются главная форма проекта (список Main form), а также автоматически создаваемые формы (Auto-create forms) и доступные проекту формы (Available forms). Кнопки в центре этой вкладки позволяют переносить формы из одного списка в другой. Флажок Default разрешает или запрещает использовать текущие параметры как значения по умолчанию для других проектов.

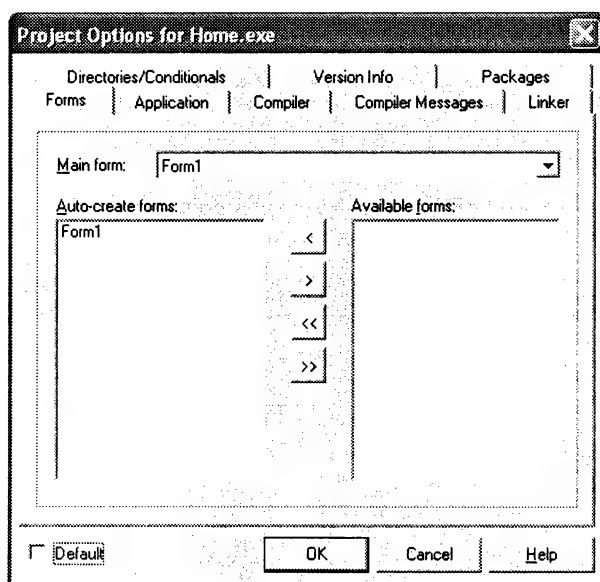


Рис. А.12. Окно параметров проекта

На вкладке Application указывается подпись на значке свернутой программы (Title), сам значок (Icon) и имя файла помощи (Help file).

На вкладке Compiler собраны флажки, управляющие параметрами процесса компиляции. В том числе (в скобках указана соответствующая директива компилятора):

- Optimizations (\$O) — включает режим оптимизации;
- Aligned record fields (\$A) — размещает данные, выравнивая их на границу 32-разрядного слова;
- Stack frames (\$W) — заставляет компилятор создавать стековые рамки для всех процедур и функций;

- Pentium-Safe FDIV (\$U) — вырабатывает код, предохраняющий от ошибок в вещественных вычислениях на процессорах Pentium ранних выпусков;
 - Range Checking (\$R) — создает код проверки выхода за границы массивов;
 - I/O Checking (\$I) — создает код проверки корректности выполнения операций ввода-вывода;
 - Overflow Checking (\$Q) — вырабатывает код проверки переполнения при выполнении целочисленных операций;
 - Strict Var-Strings (\$V) — определяет строгую проверку соответствия строковых типов при обращении к подпрограммам;
 - Complete Boolean Eval (\$B) — определяет полное вычисление логических выражений;
 - Extended Syntax (\$X) — включает расширенный синтаксис Object Pascal;
 - Typed @ Operator (\$T) — контролирует типы указателей в операции @;
 - Open Parameters (\$P) — разрешает использование открытых параметров в подпрограммах;
 - Huge Strings (\$H) — связывает зарезервированное слово **String** с длинными строками;
 - Assignable Typed Constants (\$J) — разрешает присваивание типизированным константам;
 - Debug Information (\$D) — помещает в DCU-файл отладочную информацию;
 - Local Symbols (\$L) — создает отладочную информацию о локальных символах программы;
 - Symbol Information (\$Y) — создает отладочную информацию о символах программы;
 - Show Hints — заставляет компилятор выдавать рекомендации;
 - Show Warnings — заставляет компилятор выдавать предупреждающие сообщения;
 - Assertions (\$C) — заставляет вырабатывать код для отладочных процедур.
- Вкладка **Linker** определяет параметры компоновщика, в том числе:
- Off — запрещает создавать карту распределения памяти;
 - Segments — карта содержит список модулей и адреса точек входа всех подпрограмм;
 - Publics — дополняет вариант Segments отсортированным списком символов секций **public**;
 - Detailed — дополняет вариант Publics детальной информацией о модулях;
 - Generate DCUs — создает стандартные для Delphi DCU-файлы;
 - Generate C Object files — создает файлы в формате объектов языка Си;
 - Generate C++ Object files — создает файлы для связывания с программой, созданной с помощью C++ Builder;
 - Generate Console Application — создает консольную программу;

- Include TD32 Debug Info — помещает в исполняемый файл информацию для внешнего отладчика;
- Include remote debug symbols — используется для удаленного вызова отладчика;
- Min Stack Size — устанавливает минимальный размер стека;
- Max Stack Size — устанавливает максимальный размер стека;
- Image Base — указывает начальный адрес для загрузки изображений (для DLL);
- EXE Descriptor — информационная строка длиной до 255 символов, которая включается в исполняемый файл (например, для объявления авторских прав).

Вкладка Directories/Conditionals позволяет задать каталоги размещения и условные символы:

- Output Directory — указывает каталог размещения исполняемого файла;
- Unit Output Directory — указывает каталог размещения DCU-файлов;
- Search Path — каталог размещения файлов с исходными текстами программы, если они не обнаружены в текущем каталоге (чтобы в любом из описанных выше полей указать несколько каталогов, они разделяются точкой с запятой);
- Debug source path — определяет каталог размещения внешнего отладчика;
- BPL output directory — определяет каталог размещения файлов компиляции пакетов;
- DCP output directory — определяет каталог размещения файлов для локализации программ;
- Conditional Defines — определяет символы для условной компиляции;
- Unit Aliases — определяет псевдонимы модулей.

На вкладке VersionInfo сосредоточены параметры управления информацией о версии программы:

- Include version information in project — если установлен этот флажок, в проект включается информация о версии программы, которую можно прочитать после щелчка правой кнопкой мыши на значке программы и выборе в контекстном меню команды Properties (Свойства);

ВНИМАНИЕ Все остальные интерфейсные элементы этой вкладки становятся доступными только после установки флажка Include version information in project.

- Module Version Number — поля Major, Minor, Release, Build определяют составной номер версии;
- Auto-increment build number — если установлен этот флажок, номер версии автоматически наращивается при каждой компиляции программы;
- Debug Build — указывает на создание отладочной версии программы;
- Pre-Release — указывает на создание некоммерческой версии программы;
- Special Build — указывает на специальную версию программы;
- Private Build — указывает на версию, не предназначенную для широкого распространения;

- ✱ DLL — создается динамически подключаемая библиотека;
- ✱ Language ID — идентификатор языка, на который рассчитана программа.

Меню Run

Команды меню Run перечислены ниже.

- ✱ Run — компилирует программу и делает ее прогон.
- ✱ Attach to Process — позволяет присоединиться в режиме отладки к одному из уже запущенных процессов на другой сетевой машине.
- ✱ Parameters — определяет командную строку для запуска вашей программы.
- ✱ Register ActiveX Servers — регистрирует ваш проект в реестре Windows. Команда доступна только для проектов ActiveX.
- ✱ Unregister ActiveX Servers — удаляет ваш проект из реестра Windows. Команда доступна только для проектов ActiveX.
- ✱ Install MTS Objects — регистрирует в вашем проекте объект MTS.
- ✱ Step Over — в отладочном режиме выполняет текущую строку кода без контроля вызываемых подпрограмм.
- ✱ Trace Into — в отладочном режиме выполняет текущую строку кода с контролем вызываемых подпрограмм.
- ✱ Trace To Next Source Line — программа выполняется до ближайшего от текущего положения курсора исполняемого оператора.
- ✱ Run To Cursor — в отладочном режиме выполняет программу и останавливается перед выполнением кода в строке с текстовым курсором.
- ✱ Run Until Return — в отладочном режиме выполняет текущую подпрограмму и останавливается.
- ✱ Show Execution Point — отображает в окне кода оператор, на котором было прервано выполнение программы.
- ✱ Program Pause — приостанавливает прогон отлаживаемой программы.
- ✱ Program Reset — прекращает прогон программы и восстанавливает режим разработки программы.
- ✱ Inspect — открывает окно проверки текущего значения.
- ✱ Evaluate/Modify — открывает окно проверки/изменения переменных.
- ✱ Add Watch — добавляет переменную или выражение в окно наблюдения.
- ✱ Add Breakpoint — добавляет точку останова.

Меню Component

Команды меню Component перечислены ниже.

- ✱ New Component — открывает окно эксперта компонентов.
- ✱ Install Component — помещает компонент в существующий или новый пакет.
- ✱ Import ActiveX Control — добавляет к проекту библиотеку типов компонентов ActiveX.

- Create Component Template — помещает шаблон в палитру компонентов.
- Install Packages — позволяет указать пакеты, необходимые на этапе конструирования и прогона программы.
- Configure Palette — вызывает диалоговое окно настройки палитры компонентов.

Меню Database

Команды меню Database перечислены ниже.

- Explore — вызывает инструмент исследования баз данных — SQL Explorer.
- SQL Monitor — вызывает инструмент запросов к базам данных — SQL Monitor.
- Form Wizard — вызывает окно эксперта форм для создания формы, отображающей наборы данных из удаленных или локальных баз данных.

Меню Tools

Команды меню Tools перечислены ниже.

- Environment Options — вызывает окно настройки параметров среды Delphi и ее инструментов.
- Editor Options — вызывает окно настройки параметров редактора Delphi.
- Debugger Options — вызывает окно настройки параметров отладчика Delphi.
- Translation Tools Options — вызывает окно настройки параметров локализации.
- Repository — показывает окно управления хранилищем объектов Delphi.
- Translation Repository — показывает окно хранилища объектов трансляции.
- Web App Debugger — позволяет выбрать отладчик для веб-приложений.
- Regenerate CORBA IDL Files — создает заново IDL-файлы для межмашинного взаимодействия по технологии CORBA.
- Configure Tools — показывает диалоговое окно редактирования меню Tools.
- Database Desktop — вызывает инструмент обслуживания баз данных — Database Desktop.
- Image Editor — вызывает окно редактора графики.
- Package Collection Editor — вызывает редактор наборов пакетов.
- XML Mapper — вызывает утилиту для отображения данных в виде XML-файла.
- Rave Designer — вызывает визуальную среду разработчика отчетов Rave Reports.

Настройка параметров среды

Диалоговое окно настройки параметров среды (рис. А.13) вызывается командой Tools ► Environment Options.

Вкладка Preferences открывает доступ к параметрам среды Delphi:

- Editor files — перед прогоном автоматически сохраняются все измененные файлы;
- Project Desktop — перед прогоном автоматически сохраняется информация о состоянии экрана;

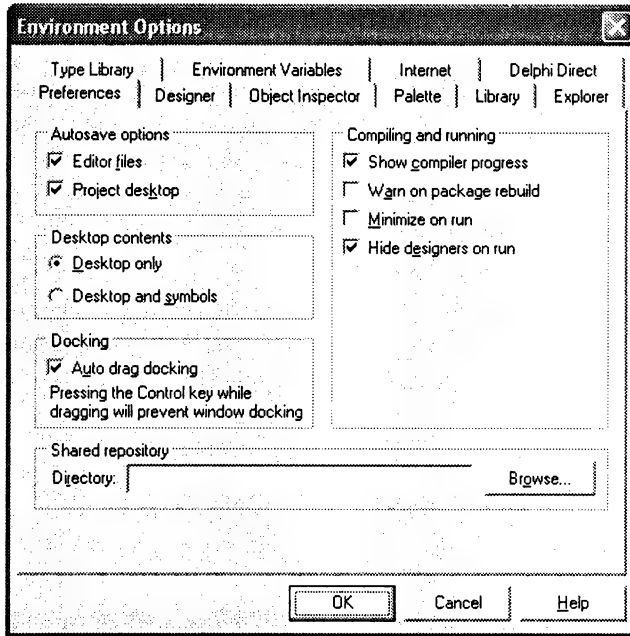


Рис. А.13. Окно настройки параметров среды

- Desktop Only — при выходе из программы сохраняется информация о состоянии экрана;
- Desktop and Symbols — при выходе из программы сохраняется информация о состоянии экрана и символах программы на момент последней удачной компиляции;
- Auto drag docking — разрешает причаливание одного инструментального окна к другому;
- Show compiler progress — показывает индикатор процесса компиляции;
- Warn on package rebuild — предупреждает о перекомпиляции пакетов;
- Minimize On Run — сворачивает окна Delphi в момент старта программы;
- Hide Designers On Run — скрывает вспомогательные окна (окно инспектора объектов и окна форм) в момент старта программы;
- Directory — путь к файлу хранилища объектов DELPHI32.DRO; если путь не указан, используется вложенный каталог BIN каталога размещения Delphi.

Вкладка Designer содержит параметры конструирования форм:

- Display Grid — показывает сетку на пустой форме;
- Snap to Grid — «привязывает» компоненты к узлам сетки;
- Show component captions — показывает имена компонентов на этапе конструирования программы;
- Show designer hints — показывает оперативную подсказку об именах компонентов и их типах на этапе конструирования;

- **Show extended control hints** — показывает оперативную подсказку с расширенной информацией;
- **New Forms as Text** — сохраняет файлы описания форм в текстовом формате;
- **Auto create forms & data modules** — при установке этого флажка каждая добавляемая к проекту форма или модуль данных помещается в список автоматически создаваемых форм, в противном случае — в список доступных форм;
- **Grid size X** — горизонтальный шаг сетки;
- **Grid size Y** — вертикальный шаг сетки.

Вкладка **Object Inspector** открывает доступ к параметрам инспектора объектов:

- **SpeedSettings** — позволяет выбрать один из следующих вариантов настройки:
 - ☐ **Custom colors & settings** (Настраиваемые цвета и установки);
 - ☐ **Default colors & settings** (Цвета и установки по умолчанию);
 - ☐ **Delphi 5 emulation** (Эмуляция Delphi 5);
 - ☐ **Visual Studio (TM) emulation** (Эмуляция Visual Studio);
- **Colors** — позволяет настроить параметры цветового выделения команд в окне инспектора объектов;
- **Show instance list** — показывает в верхней части инспектора объектов список компонентов и имен их классов;
- **Show classname in instance list** — показывает имена классов в списке инспектора объектов;
- **Show status bar** — показывает строку состояния в нижней части инспектора объектов;
- **Render background grid** — отделяет строки горизонтальными линиями;
- **Integral height (when not docked)** — если флажок установлен, изменение вертикального размера окна осуществляется так, чтобы полностью вместить значение каждого свойства или события;
- **Show read only properties** — показывает свойства, предназначенные только для чтения;
- **Bold non default values** — выделяет жирным шрифтом значения, отличающиеся от умалчиваемых;
- **Expand inline** — раскрывает свойства и методы внешнего объекта;
- **Show on events page** — показывает внешний объект на вкладке событий;

Вкладка **Palette** предоставляет средства для настройки палитры компонентов: вы можете изменять порядок следования компонентов и страниц, переименовывать страницы, добавлять к ним новые компоненты, удалять существующие и т. д.

Вкладка **Library** определяет каталоги размещения библиотек Delphi:

- **Library path** — маршрут поиска библиотечных файлов;
- **BPL output library** — выходной каталог для размещения BPL-файлов;
- **DCP output library** — выходной каталог для размещения DCP-файлов;
- **Browsing Path** — каталоги для браузера.

Вкладка Explorer позволяет настроить свойства кодового браузера, окно которого обычно «причалено» к окну редактора кода:

- ✚ Automatically show Explorer — если этот флажок установлен, окно кодового браузера будет появляться при создании каждого нового проекта;
- ✚ Highlight incomplete class items — незавершенные определения классов выделяются цветом;
- ✚ Show declaration syntax — позволяет помимо имен элементов показывать свойства и методы;
- ✚ Explorer sorting — определяет способ сортировки элементов (по алфавиту или по порядку объявления);
- ✚ Finish incomplete properties — завершает определения не только классов, но и свойств;
- ✚ Initial browser view — переключатели этой группы позволяют выбрать приоритет показа классов, модулей или глобальных определений;
- ✚ Browser scope — определяет символы, отображаемые браузером (только определенные в текущем модуле или все);
- ✚ Explorer categories — флажки этой группы определяют категории отображаемых символов.

Вкладка Type Library служит для управления свойствами редактора библиотеки типов (используется при разработке многозвенных приложений баз данных).

Вкладка Environment Variables определяет так называемые переменные среды разработки, в том числе тип операционной системы, положение основных файлов операционной системы и т. п.

Вкладка Internet управляет форматами создаваемых файлов для передачи по Интернету или локальной сети.

С помощью небольшой вкладки Delphi Direct можно управлять автоматической связью с сайтом Delphi для обновления информации об этом программном продукте.

Настройка параметров кодового редактора

Окно параметров кодового редактора вызывается командой Tools ► Editor Options. Вкладка General этого окна позволяет настроить общие свойства кодового редактора:

- ✚ Insert Mode — определяет режим вставки по умолчанию (переключается клавишей Insert);
- ✚ Cursor Through Tabs — разрешает курсору «перескакивать» через символы табуляции;
- ✚ Group Undo — разрешает удалять группу последних изменений текста при нажатии клавиш Alt+Backspace или выборе команды Edit ► Undo;
- ✚ Cursor Beyond EOF — позиция курсора включает символы конца строки;
- ✚ Double Click Line — выделяет цветом всю строку при двойном щелчке на любом ее символе;

- **Force Cut And Copy Enabled** — разрешает выполнение команд **Edit ▶ Cut** и **Edit ▶ Copy**, даже если текст не выделен;
- **Auto-complete text to find** — разрешает автоматическое дополнение текста в диалоговом окне поиска;
- **Undo After Save** — разрешает восстанавливать изменения, бывшие до последнего сохранения файла;
- **Brief Regular Expressions** — разрешает использовать шаблоны при поиске и поиске с заменой; шаблоны включают в себя следующие специальные символы:
 - **^** или **%** — в начале строки означает, что искомый фрагмент должен располагаться в начале строки;
 - **\$** или **>** — в конце строки означает, что искомый фрагмент должен располагаться в конце строки;
 - **?** — на этом месте может стоять любой символ;
 - **@** — после символа указывает на любое, в том числе нулевое, число этих символов в этом месте (**bo@** означает **boo**, **bo**, **bot**);
 - **+** — после символа указывает на любое ненулевое число символов, которые следуют дальше (**bo+** означает **boo**, **bonus**, **bot**, но не **b** или **bo**);
 - **|** — выбор одного из выражений до или после черты (**bar | car** означает **bar** или **car**);
 - **~** — на этом месте не должен стоять следующий символ;
 - **[]** — на этом месте должен стоять любой из указанных в скобках символов (**[bot]** означает **b**, **o** или **t**);
 - **[^]** — на этом месте не должен стоять любой из указанных в скобках символов (**[^bot]** означает любой символ за исключением **b**, **o** и **t**);
 - **[-]** — задает диапазон символов (**[b-o]** означает любой символ в диапазоне от **b** до **o** включительно);
 - **()** — определяет вложенный шаблон, среда допускает до 10 уровней вложенности;
 - **** — отменяет действие стоящего за ним специального символа, позволяя вести поиск этого символа как такового.
- **Persistent Blocks** — указывает, что выделенный блок остается выделенным, даже если его покидает текстовый курсор (выделение сохраняется до нового выделения);
- **Overwrite Blocks** — заменяет выделенный блок текстом из буфера обмена; если при этом установлен флажок **Persistent Blocks**, блок вставляется сразу за выделенным;
- **Find Text At Cursor** — если флажок установлен, в образце поиска при поиске или поиске с заменой будет появляться слово, рядом с которым располагается текстовый курсор;
- **Undo Limit** — определяет предельное число отменяемых командой **Edit ▶ Undo** операций;

Вкладка Source Options управляет свойствами создаваемого текста:

- Source file type — определяет тип синтаксического выделения, предусмотренный для выбранного в списке языка;
 - New — вызывает диалоговое окно для ввода нового элемента в список Source file type;
 - Delete — удаляет текущий язык в списке;
 - Auto Indent Mode — реализует автоматический отступ (при нажатии клавиши Enter курсор устанавливается на начало предыдущей строки);
 - Use Tab Character — при нажатии клавиши Tab в текст вставляется символ табуляции (если флажок не установлен, вставляются символы пробела);
 - Smart Tab — при нажатии клавиши Tab смещает курсор к первому непробельному символу в предыдущей строке;
 - Optimal Fill — начинает каждый автоматический отступ с минимального количества символов табуляции и/или пробела;
 - Backspace Unindents — разрешает удалять автоматический отступ клавишей Backspace;
 - Keep Trailing Blanks — сохраняет ведомые символы пробела;
 - Show tab character — показывает символы табуляции в виде символов >>;
 - Show space character (.) — показывает символы пробела в виде точек;
 - Use Syntax Highlighting — разрешает использование синтаксического цветовыделения;
 - Syntax Highlihghter — позволяет выбрать стиль цветовыделения;
 - Block indent — определяет отступ для блока;
 - Tab stops — определяет перемещение курсора при нажатии клавиши Tab;
 - Edit Code Templates — вызывает диалоговое окно редактирования образцов кода (см. ниже пункт «Вставка текстовых заготовок» подраздела «Интеллектуальные возможности редактора» в разделе «Работа с редактором»).
- Вкладка Display позволяет выполнить настройку экрана:
- Brief Cursor Shapes — заставляет использовать формы курсора, характерные для редактора Brief;
 - Create Backup File — заставляет создавать резервный файл с расширением, начинающимся символом тильда (~);
 - Zoom To Full Screen — разрешает разворачивание кодового окна на весь экран;
 - Sort popup pages menu — сортирует команды контекстного меню редактора кода в алфавитном порядке;
 - Visible Right Margin — разрешает показывать правую границу текста в виде вертикальной пунктирной линии;
 - Visible gutter — разрешает показывать служебную зону в левой части окна редактора;
 - Right Margin — определяет правую границу текста;

- **Gutter width** — определяет ширину левой служебной зоны;
- **Editor Font** — определяет используемый в редакторе шрифт;
- **Size** — задает высоту шрифта.

Вкладка **Color** позволяет выбрать цвета для отдельных синтаксических элементов. Выбранные цвета используются для синтаксического цветовыделения.

Вкладка **Key Mappings** позволяет настроить «горячие» клавиши, используемые в кодовом редакторе для ускорения решения типовых задач.

Вкладка **Code Insight** позволяет настроить некоторые интеллектуальные возможности кодового редактора:

- **Code Completion** — разрешает использовать подсказку в виде списка свойств, методов и событий, появляющуюся после ввода имени класса (или имени объекта) и следующей за ним точки (рис. А.14);

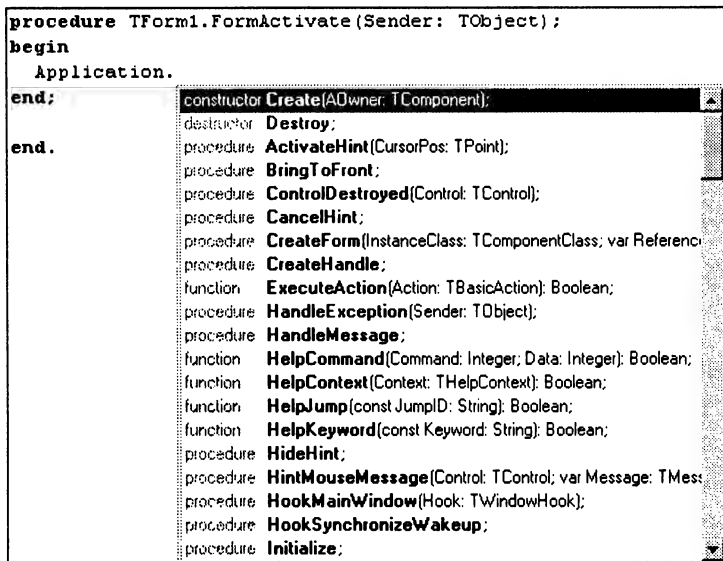


Рис. А.14. Список имен методов и свойств

- **Code Parameters** — разрешает появление подсказки с перечислением формальных параметров при определении вызовов методов класса (рис. А.15);

```
ApplyUpdates (
    const DataSets: array of TDBDataSet
```

Рис. А.15. Всплывающая подсказка с именами формальных параметров

- **Tooltip Expression Evaluation** — разрешает показывать подсказку, в которой на этапе останова программы в контрольной точке будет отображаться текущее содержимое переменной или выражения при указании на них мышью (рис. А.16);
- **Tooltip Symbol Insight** — разрешает показывать информацию о параметрах идентификатора (рис. А.17);

```

if Cashe then
begin
  Cashe = True
end

```

Рис. А.16. Всплывающая подсказка со значением переменной

```

procedure TForm1.ButtonApplyClick
begin
  DataModuleCache.DBaseCache.Apply;
end
var CUPDM.DataModuleCache - CUPDM.pas (38)

```

Рис. А.17. Всплывающая подсказка с параметрами идентификатора

- Code Completion Delay — определяет задержку в секундах до включения интеллектуальных возможностей редактора;
- CoheInsight Colors — определяет цветовыделение.

Меню Help

Команды меню Help перечислены ниже.

- Delphi Help — основная справочная служба Delphi.
- Delphi Tools — справочная служба по инструментам Delphi.
- Windows SDK — справочная служба по Windows API.
- Borland Home Page — домашняя страничка компании Borland (рис. А.18).
- Borland Developer Network — страничка поддержки разработчика Borland (рис. А.19)
- Delphi Home Page — домашняя страничка Delphi (рис. А.20).
- Borland Developer Support — страничка поддержки разработчика Delphi (рис. А.21).
- Delphi Direct — окно Интернет-поддержки разработчика (рис. А.22).
- Customize — вызов службы OpenHelp.
- About — окно с краткой информацией о Delphi.

Особенности справочной службы Delphi

Многие компоненты Delphi 7 могут использоваться для программ, работающих как под управлением Windows, так и под управлением Linux. В силу принципиальной разницы этих платформ компоненты CLX (так называется библиотека визуальных компонентов, поддерживаемая обеими платформами) имеют в общем случае несколько иной набор методов, свойств и событий, чем аналогичные компоненты VCL (то есть рассчитанные на работу только под Windows). В связи с этим при обращении к встроенной справочной службе среда показывает промежуточное диалоговое окно, в котором название компонента (или его свойства, метода и события) повторено дважды. Учтите, что верхняя строка открывает доступ к справке CLX, а нижняя — к справке VCL.

Служба OpenHelp

Служба OpenHelp предназначена для модификации справочной службы Delphi: с ее помощью можно удалять ненужные разделы и, что наиболее важно, вставлять



Рис. А. 18. Домашняя страничка компании Borland

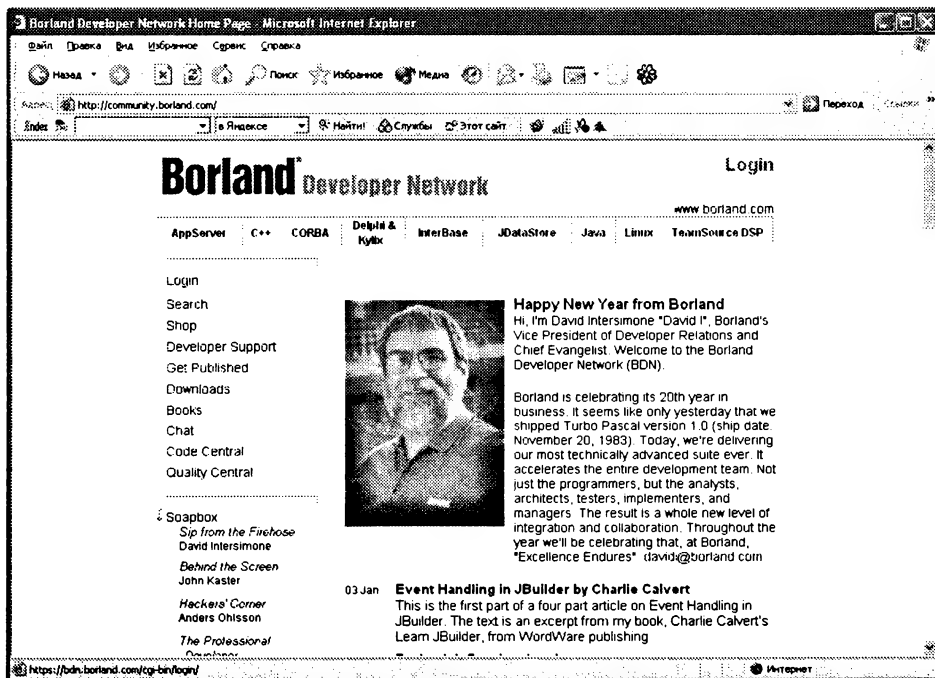


Рис. А. 19. Страничка поддержки разработчика Borland



Рис. А.20. Домашняя страничка Delphi

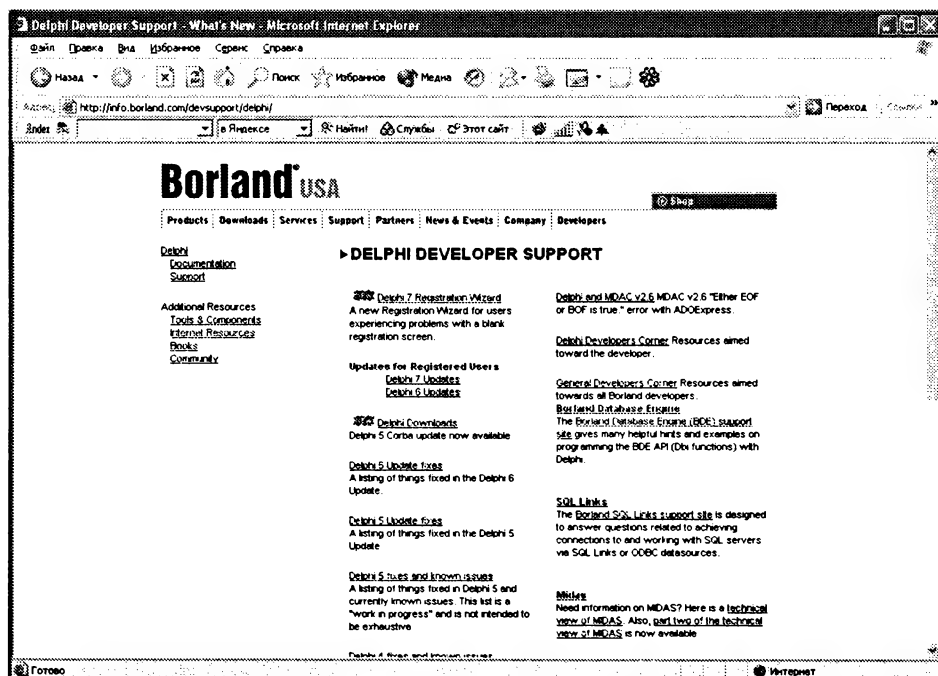


Рис. А.21. Страничка поддержки разработчика Delphi

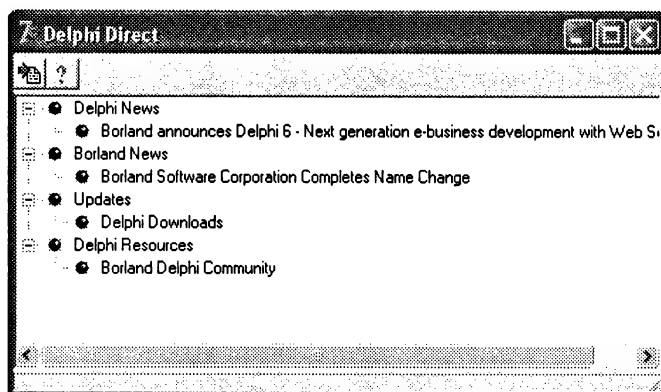


Рис. А.22. Окно Интернет-поддержки разработчика

новые. Последнее связано с тем, что существует множество относительно небольших фирм, занимающихся разработкой и продажей пакетов компонентов для Delphi и C++ Builder (пакеты компонентов для этих двух инструментов идентичны), а также компонентов ActiveX. Сама среда Delphi имеет развитые средства создания новых компонентов. Поставщики компонентов обычно поставляют вместе с ними нужные файлы справочной информации, которые OpenHelp может сделать частью справочной службы Delphi.

Служба OpenHelp вызывается командой Help ► Customize главного меню. Ее окно показано на рис. А.23.

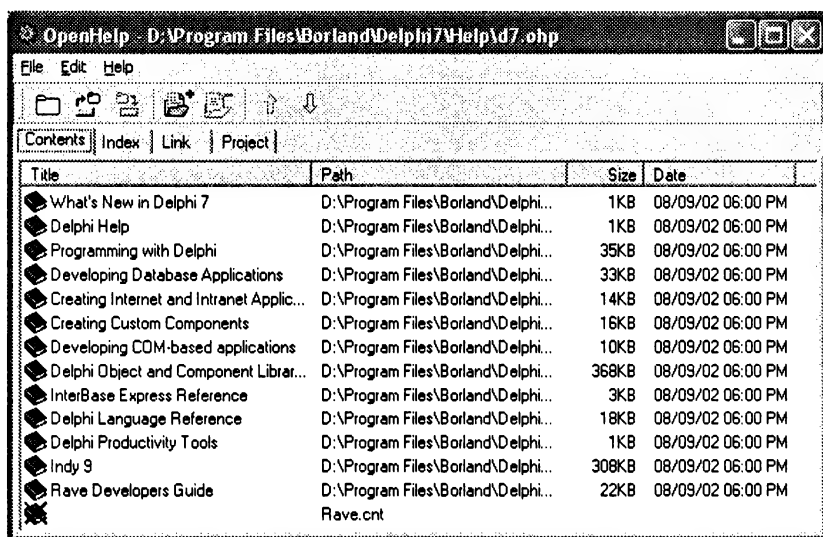


Рис. А.23. Окно службы OpenHelp

Четыре вкладки этого окна управляют содержанием справочной службы (вкладка Content), набором файлов помощи (Index), контекстной помощью (Link) и про-

ектами (Project). Чтобы удалить раздел содержания, файл справки или проект, нужно перейти на соответствующую вкладку, щелкнуть на удаляемом компоненте и выбрать команду Edit ► Remove Files или щелкнуть на кнопке  панели инструментов.

Для добавления к содержанию нового раздела требуется предварительно создать два специальных файла: файл с таблицей содержания (с расширением TOC) и файл содержания (CNT). TOC-файл, как следует из документации, есть стандартный файл содержания (CNT-файл), создаваемый утилитой hsw.exe, с тем отличием, что в нем нельзя использовать секции INCLUDE. CNT-файл, в свою очередь, отличается от стандартного тем, что в нем должны быть только две директивы: BASE и TITLE, причем содержимое TITLE у обоих файлов должно быть одинаковым, и они должны располагаться в одной папке. После создания файлов вызывается служба OpenHelp и на вкладке Content выполняется щелчок на кнопке  или выбирается команда Edit ► Add Files. В появляющемся вслед за этим диалоговым окне нужно указать путь к TOC-файлу.

Для добавления справочного файла (HLP) открывается вкладка Index, выполняется щелчок на той же кнопке и указывается путь к HLP-файлу.

Как показывает практика, эти два действия вполне достаточны для нормального подключения к справочной службе новой информации. Если вы закроете окно службы OpenHelp, появится запрос о необходимости обработки внесенных изменений. После утвердительного ответа новый вариант справочной службы будет готов к работе.

Работа с редактором

Все команды редактора можно разделить на команды перемещения курсора, команды удаления и вставки, команды работы с блоками, прочие команды.

Команды перемещения курсора

Клавиши, соответствующие командам перемещения курсора, представлены в табл. А.1.

Таблица А.1. Клавиши перемещения курсора

Клавиши	Переход	Клавиши	Переход
←	На символ влево	Home	В начало строки
→	На символ вправо	End	В конец строки
Ctrl+←	На слово влево	Ctrl+PgUp	В начало экрана
Ctrl+→	На слово вправо	Ctrl+PgDn	В конец экрана
↑	На строку вверх	Ctrl+Home	В начало файла
↓	На строку вниз	Ctrl+End	В конец файла
PgUp	На страницу вверх	Ctrl+Q+B	В начало блока
PgDn	На страницу вниз	Ctrl+Q+K	В конец блока

Команды удаления и вставки

Клавиши, соответствующие командам удаления и вставки, представлены в табл. А.2.

Таблица А.2. Клавиши для удаления и вставки

Клавиши	Описание	Клавиши	Описание
INS	Включить/отключить режим вставки	Delete	Удалить символ справа от курсора
Enter	Вставить строку	Ctrl+T	Удалить слово справа от курсора
Ctrl+Y	Удалить строку	Ctrl+Q+Y	Удалить остаток строки справа от курсора
Backspace	Удалить символ слева от курсора	Ctrl+Z	Отменить последнее изменение текста

Команды для работы с блоками

При подготовке текстов программ часто возникает необходимость перенести фрагмент текста в другое место или удалить его. Для такого рода операций удобно использовать блоки — фрагменты текста, рассматриваемые как единое целое. Длина блока может быть достаточно большой, он может занимать несколько экранных страниц. В каждый момент в одном окне редактора может быть объявлен только один блок. Обмен блоками между окнами возможен только через буфер обмена. Клавиши, соответствующие командам для работы с блоками, представлены в табл. А.3.

Таблица А.3. Клавиши для работы с блоками

Клавиши	Описание
Ctrl+K+T	Пометить в качестве блока слово слева от курсора
Ctrl+K+P	Напечатать блок
Ctrl+K+N	Убрать выделение блока цветом; повторное нажатие клавиш Ctrl+K+N вновь выделит блок
Ctrl+K+Y	Удалить блок
Ctrl+K+R	Считать блок из дискового файла в позицию, определяемую текстовым курсором
Ctrl+K+W	Записать блок на диск
Ctrl+K+I	Сместить блок вправо на два символа
Ctrl+K+U	Сместить блок влево на два символа
Shift+Delete	Вырезать блок и поместить его в буфер обмена
Ctrl+Insert	Копировать блок в буфер обмена
Shift+Insert	Вставить содержимое буфера обмена в позицию, идентифицируемую текстовым курсором

Прочие команды

В табл. А.4 представлены клавиши, соответствующие командам, не относящимся ни к одной из перечисленных выше категорий.

Таблица А.4. Клавиши для выполнения дополнительных команд

Клавиши	Описание
Ctrl+F	Искать по образцу
F3	Продолжить поиск
Ctrl+R	Искать по образцу и заменить
Ctrl+K+n	Установить маркер; $n = 0-9$ (см. ниже)
Ctrl+Q+n	Искать маркер
Ctrl+Q+]	Искать парную скобку (см. ниже)
Ctrl+O+O	Вставить параметры компилятора в начало файла (см. ниже)
Ctrl+Shift+R	Начинает и заканчивает определение макроса
Ctrl+Shift+P	Выполнить ранее определенный макрос

Хотя назначение большинства команд понятно, некоторые нуждаются в пояснении.

Нажатие клавиш Ctrl+K+n устанавливает в текущую позицию курсора маркер с номером $n = 0-9$. Маркер на экране появляется в виде небольшого окна с номером маркера в служебном поле слева от текста. Он никак не влияет на исполнение программы. После назначения маркера нажатием клавиш Ctrl+Q+n можно вызвать команду поиска маркера с номером n . Это делается для ускорения поиска нужных фрагментов текста при разработке больших программ. Раз установленный маркер нельзя удалить, но можно его поместить в другое место файла. При записи на диск маркеры не запоминаются, то есть после чтения файла с диска маркеры в нем отсутствуют.

Нажатие клавиш Ctrl+Q+] инициирует поиск ближайшей парной скобки. Таким образом удастся отыскивать пары скобок (и), { и }, [и]. Расположите курсор непосредственно перед одной из скобок и нажмите клавиши Ctrl+Q+] — редактор отыщет нужную парную скобку.

Нажатие клавиш Ctrl+O+O заставит редактор поместить в самое начало файла строки, содержащие текущие параметры среды в виде директив компилятора, например:

```
{ $A+, B-, C+, D+, E-, F-, G+, H+, I+, J+, K-, L+, M-, N+, O+, P+, Q-, R-, S-,
T-, U-, V+, W-, X+, Y- }
{ $MINSTACKSIZE $00004000 }
{ $MAXSTACKSIZE $00100000 }
{ $IMAGEBASE $00400000 }
{ $APPTYPE GUI }
```

В окне кода можно запрограммировать часто повторяющиеся манипуляции с клавишами в виде макроса. Чтобы начать запись макроса, нажимаются клавиши Ctrl+Shift+R. После этого любые действия программиста с клавиатурой запомина-

ются вплоть до повторного нажатия той же комбинации клавиш. Запомненный макрос исполняется нажатием клавиш Ctrl+Shift+P.

Интеллектуальные возможности редактора

Термин «интеллектуальные возможности» носит условный характер — он обозначает действия, которые редактор выполняет автоматически или по вашей команде и которые существенно упрощают анализ текстов программ, подготовку стандартных программных заготовок, отладку программ.

Поиск объявлений

Если активизировать окно кода и перемещать в нем указатель мыши при нажатой и удерживаемой клавише Ctrl, текст программы приобретает свойства гипертекста: на идентификаторах стандартный указатель мыши заменяется рукой с пальцем, а соответствующий идентификатор выделяется цветом и подчеркивается. Если в этот момент нажать левую кнопку мыши, редактор попытается отыскать исходный текст модуля, в котором объявлен соответствующий тип, подпрограмма или глобальная переменная и, если поиск окажется удачным, загрузит текст модуля в окно кода и установит в нем текстовый курсор в начале описания типа (подпрограммы, переменной). Того же эффекта можно достичь, если щелкнуть на идентификаторе правой кнопкой мыши и выбрать в контекстном меню команду Find Declaration.

Поиск идет в следующем порядке (перечислены команды меню и поля соответствующих окон, содержащие пути к нужным каталогам):

1. Project ► Options ► Directories ► Conditionals ► Search path.
2. Project ► Options ► Directories ► Conditionals ► Debug source path.
3. Tools ► Environment Options ► Library ► Browsing path.
4. Tools ► Environment Options ► Library ► Library path.

Нельзя искать объявления, если текущий программный модуль еще ни разу не был сохранен на диске.

Если мышь оказывается на идентификаторе без нажатой и удерживаемой клавиши Shift, рядом с указателем мыши появляется всплывающая (оперативная) подсказка, в которой сообщается, к какому элементу языка относится идентификатор (к процедуре, функции, переменной и т. д.), а также имя модуля и номер строки в нем, где этот идентификатор впервые описан.

Создание стандартных заготовок для новых свойств и методов

При объявлении новых свойств класса в интерфейсной секции вы можете написать лишь имя свойства и его тип. После нажатия клавиш Ctrl+Shift+C или щелчка правой кнопкой мыши и выбора в контекстном меню команды Complete Class at Cursor редактор добавит необходимые элементы **Read** и **Write** в описание свойства и внесет другие изменения в текст программы. Пусть, например, вы написали следующий код и нажали клавиши Ctrl+Shift+C (текстовый курсор при этом должен находиться в любом месте внутри описания класса):

```

type TMyButton = class(TButton)
    property Size: Integer;
    procedure DoSomething;
end;

```

Редактор изменит описание класса следующим образом:

```

type TMyButton = class(TButton)
    property Size: Integer read FSize write SetSize;
    procedure DoSomething;
private
    FSize: Integer;
    procedure SetSize(const Value: Integer);
end;

```

Кроме того, в исполняемую секцию будет добавлено описание двух методов:

```

{ TMyButton }
procedure TMyButton.DoSomething;
begin
    |
end;

procedure TMyButton.SetSize(const Value: Integer);
begin
    FSize := Value;
end;

```

Вы можете также вставить в раздел **implementation** новый метод класса и нажать клавиши Ctrl+Shift+C — редактор вставит прототип метода в объявление класса в секции **interface**.

Навигация внутри модуля

Клавиши перемещения курсора ↑ и ↓ в сочетании с нажатыми и удерживаемыми клавишами Ctrl и Shift осуществляют переключение между секциями **interface** и **implementation** текущего модуля. Если в секции **interface** вас заинтересовала реализация того или иного метода, щелкните на нем мышью и нажмите клавиши Ctrl+Shift+↓ — редактор отыщет реализацию и покажет ее вам. Наоборот, чтобы из секции **implementation** переместиться к заголовку метода в секции **interface**, нажмите клавиши Ctrl+Shift+↑.

Вставка текстовых заготовок

Редактор может вставлять в текст множество текстовых заготовок, позволяющих экономить время ввода кода программы. Просмотреть имеющиеся текстовые заготовки и при необходимости добавить к ним собственные можно с помощью кнопки Edit Code Templates, расположенной на вкладке Source Options окна Editor Properties, открываемого командой Tools ► Editor Options. Окно редактирования текстовых заготовок показано на рис. А.24.

В списке Templates можно выбрать имя и краткое описание образца, а в поле Code увидеть и при желании отредактировать соответствующую текстовую заго-

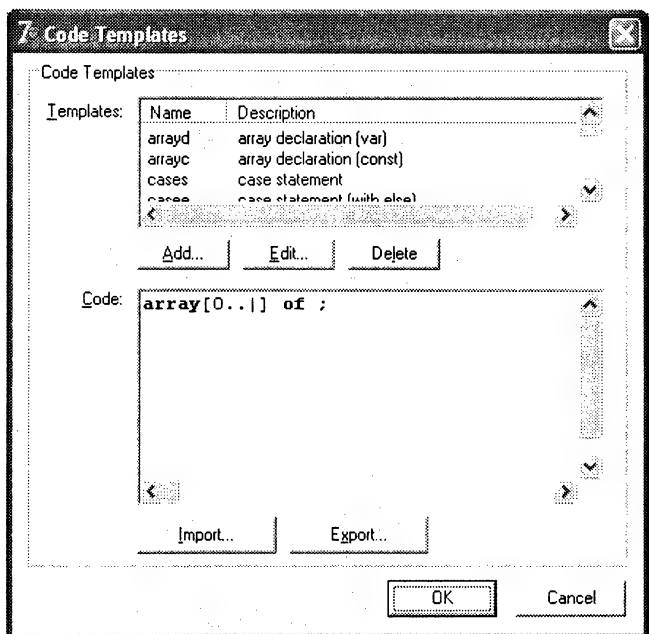


Рис. А.24. Окно Code Templates

товку. Кнопка Add позволяет добавить новый образец, кнопка Edit — изменить его имя и краткое описание, а кнопка Delete — удалить. При редактировании имеющейся или вставки новой заготовки учтите, что символ вертикальной черты (|) определяет позицию текстового курсора после вставки заготовки в текст программы.

Для вставки созданной заготовки введите ее имя в окне кода и нажмите клавиши Ctrl+J — имя заменится полным текстом заготовки. Если вы не помните всех имен заготовок, их можно выбирать в списке. Для этого установите текстовый курсор в то место, в которое вы хотите вставить заготовку, и нажмите клавиши Ctrl+J — на экране появится окно с именами и краткими описаниями всех заготовок.

Отладка программ

В Delphi имеется мощный встроенный отладчик, значительно упрощающий отладку программ. Основными инструментами отладки являются точки контрольного останова и окно наблюдения за переменными.

Точки контрольного останова

Точка контрольного останова определяет оператор в программе, перед выполнением которого программа прервет свою работу, и управление будет передано среде Delphi. Точка останова задается командой View ► Debug windows ► Breakpoints.

Окно точек останова (рис. А.25) содержит список всех установленных в проекте точек, перед обработкой которых происходит прекращение работы программы и управление получает среда Delphi.

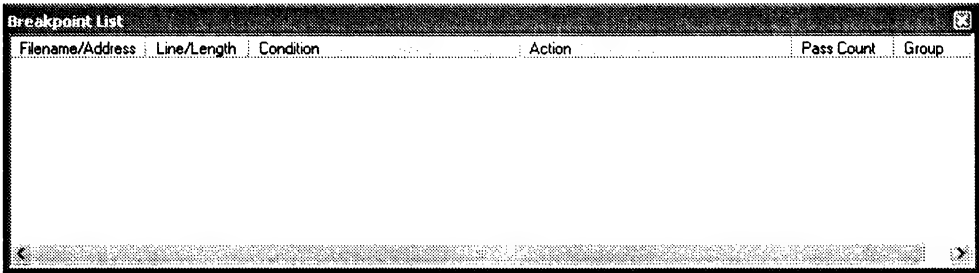


Рис. А.25. Пустое окно точек останова

Для добавления новой точки следует щелкнуть в окне правой кнопкой мыши и выбрать в контекстном меню команду **Add ► Source Breakpoint**. В этом случае появляется окно, с помощью которого можно задать параметры добавляемой точки (рис. А.26):

- **Filename** — путь и имя файла (в момент появления окна в поле этого списка содержатся данные открытого файла);
- **Line number** — номер строки от начала файла (в момент появления окна в поле этого списка содержится номер строки с текстовым курсором);
- **Condition** — в поле этого списка можно указать условие останова в виде логического выражения (например, `MyValue = MaxValue-12`);
- **Pass count** — в поле этого списка можно указать количество проходов программы через контрольную точку без прерывания вычислений;

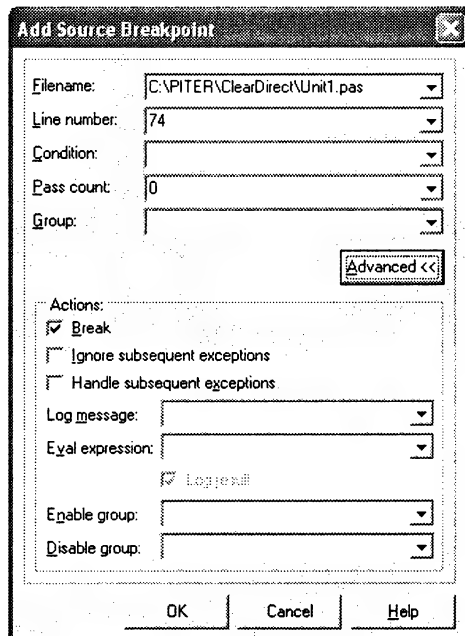


Рис. А.26. Окно добавления новой точки

- **Group** — определяет группу, к которой принадлежит точка останова. Описание раскрывающегося списка Group см. ниже в подразделе «Группировка точек прерывания».
- **Advanced** — щелчок на кнопке открывает или закрывает нижнюю часть окна Add Source Breakpoint.

Окно наблюдения

Наблюдать за состоянием переменной или выражения можно с помощью специального окна (рис. А.27), вызываемого командой View ► Debug windows ► Watches.

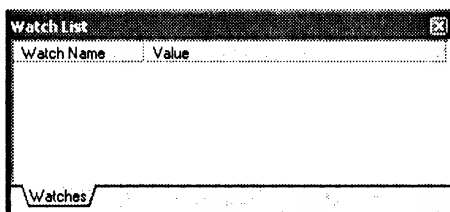


Рис. А.27. Пустое окно наблюдения

Окно наблюдения используется в отладочном режиме для наблюдения за изменением значений выражений, помещенных в это окно. Для добавления нового выражения щелкните в окне наблюдения правой кнопкой мыши и выберите в контекстном меню команду Add Watch. В поле раскрывающегося списка Expression открывшегося окна (рис. А.28) введите выражение. Подобно точкам останова, объекты наблюдения можно группировать. Для этого используется список Group name. Поле Repeat count определяет количество показываемых элементов массивов данных; поле Digits — количество значащих цифр для отображения вещественных данных; флажок Enabled разрешает или запрещает вычисление выражения. Флажок Allow Function Calls разрешает вычислять и показывать объекты, которые определяются в нетрассируемых подпрограммах. Остальные элементы определяют способ представления значения. Замечу, что вы можете просмотреть в отладочном режиме текущее значение любой переменной, если задержите на ней указатель мыши: значение появится во всплывающей подсказке рядом с указателем.

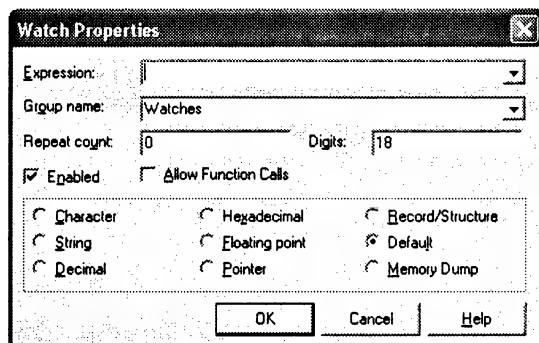


Рис. А.28. Окно добавления в окно наблюдения нового выражения

Принудительное прерывание работы программы

Если программа запущена из среды Delphi, ее работу можно прервать в любой момент с помощью клавиш **Ctrl+F2**, кнопки , команды **Run ► Program pause** или, наконец, установив точку контрольного останова в той части программы, которая выполняется в данный момент или будет выполнена в дальнейшем.

Трассировка программы

Перед исполнением оператора, в котором установлена точка контрольного останова, работа программы будет прервана, управление получит среда Delphi, а в окне наблюдения отразится текущее значение наблюдаемых переменных и/или выражений. Теперь программист может проследить работу программы по шагам с помощью клавиш **F7** и **F8** или кнопок  и  панели инструментов. При нажатии клавиши **F8** будут выполнены запрограммированные в текущей строке действия, и работа программы прервется перед выполнением следующей строки текста. Замечу, что контрольная точка останова выделяется по умолчанию красным цветом, а текущая прослеживаемая строка — синим. Если программа остановлена в контрольной точке, когда текущая строка совпадает со строкой останова, строка выделяется красным цветом. Признаком текущей строки является особый символ выделения строки в служебной зоне слева в окне редактора (рис. А.29).

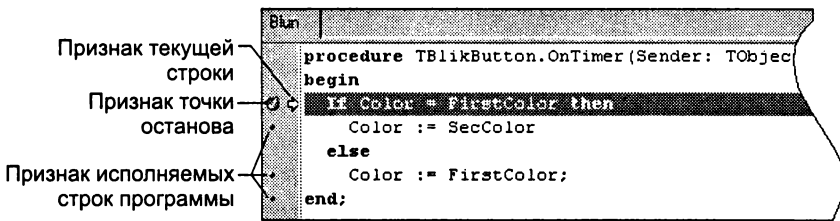


Рис. А.29. Фрагмент окна редактора в режиме отладки

Кстати, чтобы установить/снять точку контрольного останова, достаточно щелкнуть мышью в служебной зоне слева от нужной строки или установить в эту строку текстовый курсор и нажать клавишу **F5**.

При нажатии клавиши **F7** среда выполняет те же действия, что и при нажатии клавиши **F8**, однако если в текущей строке содержится вызов подпрограммы пользователя, программа прервет свою работу перед выполнением первого исполняемого оператора *этой подпрограммы*, то есть клавиша **F7** позволяет проследить работу вызываемых подпрограмм.

После трассировки нужного фрагмента программы можно продолжить нормальную ее работу, нажав клавишу **F9**.

Действия в точках прерывания

С любой точкой можно связать одно или несколько действий. Для этого нужно щелкнуть на точке останова правой кнопкой мыши и выбрать команду **Properties**. В появившемся окне свойств точек останова остается щелкнуть на кнопке **Advanced**, чтобы получить доступ к дополнительным свойствам (рис. А.30).

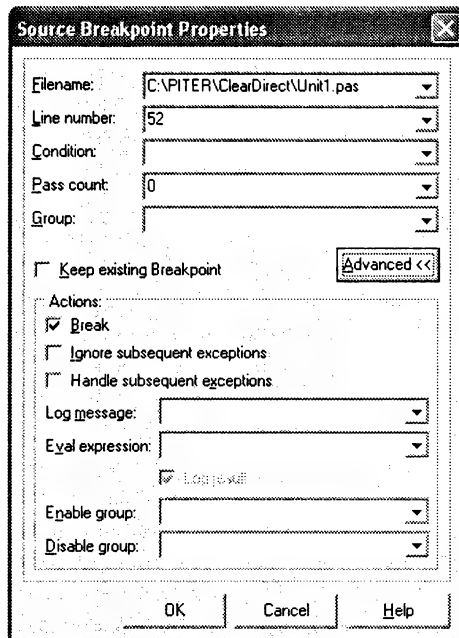


Рис. А.30. Окно связывания точки останова с действием

В нижней части окна имеется область Actions, с помощью которой и определяются действия для точки останова, указанной в верхней части окна.

- Break — простой останов перед выполнением помеченного оператора.
- Ignore subsequent exceptions — если флажок установлен, игнорируются все возможные последующие исключения в текущем отладочном сеансе до очередной точки останова, в которой, возможно, это действие будет отменено.
- Handle subsequent exceptions — после установки этого флажка отменяется действие предыдущего флажка и возобновляется обработка возможных исключений.
- Log message — этот список позволяет выбрать произвольное сообщение, связанное с точкой останова.
- Eval expression — этот список позволяет вычислить некоторое выражение и поместить его результат в сообщение, выбранное в списке Log message.

Описание раскрывающихся списков Enable group и Disable group см. в следующем подразделе.

Группировка точек прерывания

В Delphi имеется возможность объединения точек останова в группы. Для этого используется все то же окно (см. рис. А.30): в раскрывающемся списке Group следует выбрать имя группы, к которой принадлежит точка, а с помощью списков Enable Group и Disable Group соответственно разрешить или запретить действие всех точек останова, относящихся к выбранной группе.

Вычисление выражений и изменение значений

С помощью окна Evaluate/Modify (рис. А.31) можно узнать значение любого выражения или установить другое значение переменной. Это окно вызывается в режиме отладки нажатием клавиш Ctrl+F7.

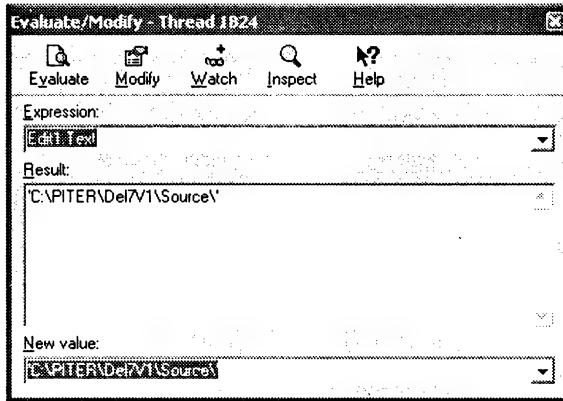


Рис. А.31. Окно прослеживания/изменения значений

Это окно — модальное, то есть оно прерывает отладку программы до тех пор, пока не будет закрыто. В поле списка Expression можно ввести имя переменной или интересующее вас выражение. После щелчка по кнопке Evaluate в поле Result появится текущее значение переменной (выражения). Если в поле Expression содержится имя переменной, одновременно становится доступной кнопка Modify, а в поле списка New value повторяется текущее значение переменной. Если изменить это значение и щелкнуть на кнопке Modify, переменная получит новое значение, которое и будет использоваться при дальнейшем прогоне программы (если определяется значение выражения, кнопка Modify и поле New value будут недоступными).

Если вы щелкнете на кнопке Watch, выражение (переменная) из окна Evaluate будет перенесено в окно наблюдений Watch, щелчок на кнопке Inspect отображает выражение (переменную) в специальном окне Debug Inspector (рис. А.32).

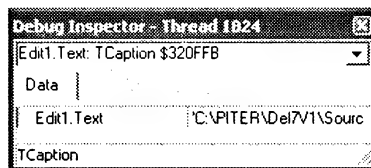


Рис. А.32. Окно инспектора отладки

ПРИМЕЧАНИЕ

Как уже упоминалось в подразделе «Интеллектуальные возможности редактора» раздела «Работа с редактором», при перемещении указателя мыши в режиме отладки на имени переменной текущее значение этой переменной появляется во всплывающей подсказке рядом с указателем.

Ведение протокола работы программы

В ряде случаев бывает неудобно или невозможно пользоваться пошаговой отладкой программ. Если вы, например, установите точку останова в подпрограмме прорисовки сетки `TDBGGrid`, программа после останова не сможет нормально продолжить свою работу, так как в этом случае она будет пытаться восстановить экран и вновь будет остановлена и т. д. В таких ситуациях вам могут помочь контрольные точки, которые не прерывают работу программы, а лишь помещают некоторую информацию в специальный файл трассировки. Для реализации такой точки раскройте окно **Source Breakpoint Properties** (см. рис. А.30), снимите флажок **Break** и введите сообщение в поле раскрывающегося списка **Log message**. Вы можете также в поле списка **Eval expression** ввести некоторое выражение, которое будет вычислено и вместе с сообщением помещено в протокол работы программы. Этот протокол можно просмотреть в любой момент (в том числе и после завершения прогона программы) с помощью команды **View ► Debug Windows ► Event Log**.

Приложение Б

Некоторые стандартные подпрограммы, переменные, классы

В этом приложении приводятся некоторые стандартные подпрограммы, системные переменные и классы, определенные в различных программных модулях.

Подпрограммы модуля Match

Подпрограмма	Описание
<i>Тригонометрические функции</i>	
function ArcCos(X: Extended): Extended;	Арккосинус
function ArcCot(const X: Extended): Extended;	Арккотангес
function ArcSin(X: Extended): Extended;	Арксинус
function ArcTan2(Y, X: Extended): Extended;	Вычисляет арктангенс Y/X и возвращает угол в правильном квадранте (функция ArcTan модуля System не учитывает квадрант)
function Cos(X: Extended): Extended;	Косинус
function Cosecant(const X: Extended): Extended;	Косеканс (1/Sin(X))
function Cotan(X: Extended): Extended;	Котангенс
function Hypot(X, Y: Extended): Extended;	Корень квадратный из (X ² + Y ²) – гипотенуза прямоугольного треугольника по двум катетам

продолжение ➤

Подпрограмма	Описание
function Sec(const X: Extended): Extended;	Секанс ($1/\cos(X)$)
function Sin(X: Extended): Extended;	Синус
procedure SinCos(Theta: Extended; var Sin, Cos: Extended);	Возвращает одновременно синус и косинус угла Theta (почти в 2 раза быстрее, чем раздельное получение синуса и косинуса)
function Tan(X: Extended): Extended;	Тангенс
<i>Функции преобразования углов</i>	
function CycleToRad(Cycles: Extended): Extended;	$\text{Radians} := \text{Cycles} \times 2\pi$
function DegToRad(Degrees: Extended): Extended;	$\text{Radians} := \text{Degrees} \times \pi/180$
function GradToRad(Grads: Extended): Extended;	$\text{Radians} := \text{Grads} \times \pi/200$
function RadToDeg(Radians: Extended): Extended;	$\text{Degrees} := \text{Radians} \times 180/\pi$
function RadToGrad(Radians: Extended): Extended;	$\text{Grads} := \text{Radians} \times 200/\pi$
function RadToCycle(Radians: Extended): Extended;	$\text{Cycles} := \text{Radians}/2\pi$
<i>Гиперболические функции</i>	
function ArcCosh(X: Extended): Extended;	Гиперболический арккосинус
function ArcSinh(X: Extended): Extended;	Гиперболический арксинус
function ArcTanh(X: Extended): Extended;	Гиперболический арктангенс
function Cosh(X: Extended): Extended;	Гиперболический косинус
function Sinh(X: Extended): Extended;	Гиперболический синус
function Tanh(X: Extended): Extended;	Гиперболический тангенс
<i>Логарифмические функции</i>	
function LnXP1(X: Extended): Extended;	Логарифм натуральный от $(X+1)$. Используется, когда значение X близко к нулю
function Log10(X: Extended): Extended;	Десятичный логарифм

Подпрограмма	Описание
function Log2(X: Extended): Extended;	Двоичный логарифм
function LogN(Base, X: Extended): Extended;	Логарифм от X при основании Base
<i>Экспоненциальные функции</i>	
function IntPower(Base: Extended; Exponent: Integer): Extended;	Возведение Base в целочисленную степень Exponent
function Power(Base, Exponent: Extended): Extended;	Возведение Base в вещественную степень Exponent
<i>Подпрограммы разного назначения</i>	
function Ceil(X: Extended): Integer;	Ближайшее меньшее целое
procedure DivMod(Dividend: Integer; Divisor: Word; var Result, Remainder: Word);	Целочисленное деление: Result – div ; Remainder – mod
function EnsureRange(const AValue, AMin, AMax: Integer): Integer;	Возвращает результат внутри диапазона AMin...AMax: если AValue в диапазоне, возвращает AValue, иначе AMin или AMax. Вместо типа Integer могут использоваться типы Int64 или Double
function Floor(X: Extended): Integer;	Ближайшее большее целое
procedure Frexp(X: Extended; var Mantissa: Extended; var Exponent: Integer);	Возвращает мантиссу и степень вещественного числа
function InRange(const AValue, AMin, AMax: Integer): Boolean;	Возвращает True, если AValue в диапазоне AMin...AMax. Вместо типа Integer могут использоваться типы Int64 или Double
function Ldexp(X: Extended; P: Integer): Extended;	Возвращает $X \times P^2$
function Poly(X: Extended; const Coefficients: array of Double): Extended;	Значение полинома $A X^N + B X^{N-1} + \dots + Z$. Коэффициенты задаются в порядке возрастания степени
<i>Статистические подпрограммы</i>	
function Max(A, B: Integer): Integer; overload ;	Возвращает максимальное из двух чисел. Вместо типа Integer могут использоваться типы Int64, Single, Double, Extended
function MaxIntValue(const Data: array of Integer): Integer;	Возвращает максимальное из набора целых чисел

Подпрограмма	Описание
function MaxValue(const Data: array of Double): Double;	Возвращает максимальное из набора вещественных чисел
function Mean(const Data: array of Double): Extended;	Вычисляет арифметическое среднее для набора вещественных чисел
procedure MeanAndStdDev(const Data: array of Double; var Mean, StdDev: Extended);	Вычисляет арифметическое среднее и стандартное отклонение для набора вещественных чисел
function Min(A,B: Integer): Integer; overload ;	Возвращает минимальное из двух чисел. Вместо типа Integer могут использоваться типы Int64, Single, Double, Extended
function MinIntValue(const Data: array of Integer): Integer;	Возвращает минимальное из набора целых чисел
function MinValue(const Data: array of Double): Double;	Возвращает минимальное из набора вещественных чисел
procedure MomentSkewKurtosis(const Data: array of Double; var M1, M2, M3, M4, Skew, Kurtosis: Extended);	Вычисляет статистические моменты порядков с первого по четвертый, а также асимметрию Skew и эксцесс Kurtosis для набора чисел
function Norm(const Data: array of Double): Extended;	Возвращает норму (квадратный корень из суммы квадратов) вещественных чисел
function PopnStdDev(const Data: array of Double): Extended;	Выборочное стандартное отклонение. Отличается от обычного стандартного отклонения тем, что использует выборочное значение дисперсии (см. ниже функцию PopnVariance)
function PopnVariance(const Data: array of Double): Extended;	Выборочная дисперсия. Использует «смещенную» формулу TotalVariance/N (см. ниже функцию TotalVariance)
function RandG(Mean, StdDev: Extended): Extended;	Генерирует псевдонормально распределенную последовательность чисел с заданным средним значением Mean и стандартным отклонением StdDev
function StdDev(const Data: array of Double): Extended;	Вычисляет среднеквадратическое отклонение для набора чисел
function Sum(const Data: array of Double): Extended register;	Вычисляет сумму чисел
procedure SumsAndSquares(const Data: array of Double): Extended;	Одновременное вычисление суммы и суммы квадратов для набора чисел
function SumInt(const Data: array of Integer): Integer register;	Сумма набора целых чисел

Подпрограмма	Описание
function SumOfSquares(const Data: array of Double): Extended;	Сумма квадратов чисел
function TotalVariance(const Data: array of Double): Extended;	Сумма квадратов расстояний всех величин от их среднего арифметического
function Variance(const Data: array of Double): Extended;	Выборочная дисперсия для набора чисел. Использует «несмещенную» формулу TotalVariance/(N-1)
<i>Финансовые функции</i>	
type TPaymentTime = (ptEndOfPeriod, ptStartOfPeriod);	Перечисленный тип, используемый в финансовых функциях
function DoubleDecliningBalance (Cost, Salvage: Extended; Life, Period: Integer): Extended;	Вычисление амортизации методом двойного баланса
function FutureValue (Rate: Extended; NPeriods: Integer; Payment, PresentValue: Extended; PaymentTime: TPaymentTime): Extended;	Будущее значение вложения
function InterestPayment (Rate: Extended; Period, NPeriods: Integer; PresentValue, FutureValue: Extended; PaymentTime: TPaymentTime): Extended;	Вычисление процентов по ссуде
function InterestRate (NPeriods: Integer; Payment, PresentValue, FutureValue: Extended; PaymentTime: TPaymentTime): Extended;	Норма прибыли, необходимая для получения заданной суммы
function InternalRateOfReturn (Guess: Extended; const CashFlows: array of Double): Extended;	Вычисление внутренней скорости оборота вложения для ряда последовательных выплат
function NetPresentValue (Rate: Extended; const CashFlows: array of Double; PaymentTime: TPaymentTime): Extended;	Вычисление чистой текущей стоимости вложения для ряда последовательных выплат с учетом процентной ставки
function NumberOfPeriods (Rate, Payment, PresentValue, FutureValue: Extended; PaymentTime: TPaymentTime): Extended;	Количество периодов, за которые вложение достигнет заданной величины
function Payment (Rate: Extended; NPeriods: Integer; PresentValue, FutureValue: Extended; PaymentTime: TPaymentTime): Extended;	Размер периодической выплаты для погашения ссуды при заданном числе периодов, процентной ставке, а также текущем и будущем значениях ссуды

Подпрограмма	Описание
function PeriodPayment (Rate: Extended; Period, NPeriods: Integer; PresentValue, FutureValue: Extended; PaymentTime: TPaymentTime): Extended;	Платежи по процентам за заданный период
function PresentValue (Rate: Extended; NPeriods: Integer; Payment, FutureValue: Extended; PaymentTime: TPaymentTime): Extended;	Текущее значение вложения
function SLNDepreciation (Cost, Salvage: Extended; Life: Integer): Extended;	Вычисление амортизации методом постоянной нормы
function SYDDepreciation (Cost, Salvage: Extended; Life, Period: Integer): Extended;	Вычисление амортизации методом весовых коэффициентов

Подпрограммы модуля DateUtils

Подпрограмма	Описание
type TValueRelationship = -1..1; function CompareDate (const A, B: TDateTime): TValueRelationship;	Сравнивает две даты
function CompareDateTime (const A, B: TDateTime): TValueRelationship;	Сравнивает две даты-времени
function CompareTime (const A, B: TDateTime): TValueRelationship;	Сравнивает два времени
function CurrentYear: Word;	Возвращает текущий год по григорианскому календарю
function DateOf (const AValue: TDateTime): TDateTime;	Возвращает дату
function DayOf (const AValue: TDateTime): Word;	Возвращает номер дня в месяце (1...31)
function DayOfTheWeek (const AValue: TDateTime): Word;	Возвращает день недели (1...7, 1 — понедельник, 7 — воскресенье)
function DayOfTheYear (const AValue: TDateTime): Word;	Возвращает порядковый номер дня в году (1...366)
function DayOfWeek (Date: TDateTime): Integer;	Возвращает день недели (1...7, 1 — воскресенье, 7 — суббота)
function DaysBetween (const ANow, AThen: TDateTime): Integer;	Возвращает полное количество дней между двумя датами
function DaysInAMonth (const AYear, AMonth: Word): Word;	Возвращает количество дней в месяце

Подпрограмма	Описание
function DaysInAYear(const AYear: Word): Word;	Возвращает количество дней в году
function DaysInMonth(const AValue: TDateTime): Word;	Возвращает количество дней в месяце
function DaysInYear(const AValue: TDateTime): Word;	Возвращает количество дней в году
function DaySpan(const ANow, AThen: TDateTime): Double;	Возвращает разницу дат, включая время
procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Word);	По дате-времени возвращает год, месяц и день месяца
procedure DecodeDateDay(const AValue: TDateTime; out AYear, ADayOfYear: Word);	По дате-времени возвращает год и день недели
function DecodeDateFully(const DateTime: TDateTime; var Year, Month, Day, DOW: Word): Boolean;	По дате-времени возвращает год, месяц, день месяца и день недели. Возвращает True, если год високосный
procedure DecodeDateMonthWeek(const AValue: TDateTime; out AYear, AMonth, AWeekOfMonth, ADayOfWeek: Word);	По дате-времени возвращает год, месяц, неделю месяца и день недели
procedure DecodeDateTime(const AValue: TDateTime; out AYear, AMonth, ADay, AHour, AMinute, ASecond, AMilliSecond: Word);	По дате-времени возвращает год, месяц, день месяца, часы, минуты, секунды и миллисекунды
procedure DecodeDateWeek(const AValue: TDateTime; out AYear, AWeekOfYear, ADayOfWeek: Word);	По дате-времени возвращает год, неделю года (1..52), день недели
procedure DecodeDayOfWeekInMonth(const AValue: TDateTime; out AYear, AMonth, ANthDayOfWeek, ADayOfWeek: Word);	По дате-времени возвращает год, месяц, неделю года (1..52), день недели
procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word);	По дате-времени возвращает часы, минуты, секунды и миллисекунды
function EncodeDate(Year, Month, Day: Word): TDateTime; function TryEncodeDate(Year, Month, Day: Word; out Date: TDateTime): Boolean;	По году, месяцу, дню формирует дату-время
function EncodeDateDay(const AYear, ADayOfYear: Word): TDateTime;	По году и номеру дня формирует дату-время
function EncodeDateMonthWeek(const AYear, AMonth, AWeekOfMonth: Word; const ADayOfWeek: Word = 1): TDateTime;	По году, месяцу, неделе месяца и дню недели формирует дату-время

Подпрограмма	Описание
function EncodeDateTime(const AYear, AMonth, ADay, AHour, AMinute, ASecond, AMilliSecond: Word): TDateTime;	По году, месяцу, дню, часу, минутам, секундам и миллисекундам формирует дату-время
function EncodeDateWeek(const AYear, AWeekOfYear: Word; const ADayOfWeek: Word = 1): TDateTime;	По году, неделе года и дню недели формирует дату-время
function EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime; function TryEncodeTime(Hour, Min, Sec, MSec: Word; out Time: TDateTime): Boolean;	По часу, минутам, секундам и миллисекундам формирует дату-время
function EndOfDay(const AYear, ADayOfYear: Word): TDateTime; overload ; function EndOfDay(const AYear, AMonth, ADay: Word): TDateTime; overload ;	Возвращает дату-время для последней миллисекунды указанной даты
function EndOfAMonth(const AYear, AMonth: Word): TDateTime;	Возвращает дату-время для последней миллисекунды последнего дня указанного месяца
function EndOfAWeek(const AYear, AWeekOfYear: Word; const ADayOfWeek: Word = 7): TDateTime;	Возвращает дату-время для последней миллисекунды последнего дня указанной недели
function EndOfAYear(const AYear): TDateTime;	Возвращает дату-время для последней миллисекунды последнего дня указанного года
function EndOfTheXXXX(const AValue: TDateTime): TDateTime;	XXXX — Day, Month, Week, Year. Возвращает дату-время для последней миллисекунды указанной даты (месяца, недели, года)
function HourOf(const AValue: TDateTime): Word;	Возвращает час указанной даты
function HourOfTheXXXX(const AValue: TDateTime): Word;	XXXX — Month, Week, Year. Возвращает количество часов между указанной датой и началом первого дня того же месяца (недели, года)
function HoursBetween(const ANow, AThen: TDateTime): Int64;	Возвращает количество часов между датами
procedure IncAMonth(var Year, Month, Day: Word; NumberOfMonths: Integer = 1);	Увеличивает указанную дату на нужное количество месяцев

Подпрограмма	Описание
function IncXXXX(const AValue: TDateTime; const ANumberOfXXXX: Integer = 1): TDateTime;	XXXX — Day, Hour, MilliSecond, Minute, Month, Second, Week, Year. Увеличивает указанную дату на нужное количество дней (часов, миллисекунд, минут, месяцев, секунд, недель, лет)
function IsInLeapYear(const AValue: TDateTime): Boolean;	Возвращает True, если указанная дата принадлежит високосному году
function IsLeapYear(Year: Word): Boolean;	Возвращает True, если указанный год високосный
function IsSameDay(const AValue, ABasis: TDateTime): Boolean;	Возвращает True, если обе даты-времени принадлежат одному дню
function IsToday(const AValue: TDateTime): Boolean;	Возвращает True, если указанная дата-время принадлежит текущему дню
function MilliSecondOf(const AValue: TDateTime): Word;	Возвращает количество миллисекунд указанного времени
function MilliSecondOfTheXXXX(const AValue: TDateTime): LongWord;	XXXX — Day, Hour, Minute, Month, Second, Week, Year. Возвращает количество миллисекунд от начала указанного дня (часа, минуты, месяца, секунды, недели, года)
function MilliSecondsBetween(const ANow, AThen: TDateTime): Int64;	Возвращает количество миллисекунд между двумя датами
function MinuteOf(const AValue: TDateTime): Word;	Возвращает количество минут указанного времени
function MinuteOfTheXXXX(const AValue: TDateTime): Word;	XXXX — Day, Hour, Month, Week, Year. Возвращает количество минут от начала указанного дня (часа, месяца, недели, года)
function MinutesBetween(const ANow, AThen: TDateTime): Int64;	Возвращает количество минут между двумя датами
function MonthOf(const AValue: TDateTime): Word;	Возвращает месяц указанной даты
function MonthsBetween(const ANow, AThen: TDateTime): Integer;	Возвращает количество месяцев между двумя датами
function RecodeDateTime(const AValue: TDateTime; const AYear, AMonth, ADay, AHour, AMinute, ASecond, AMilliSecond: Word): TDateTime; function RecodeDate(const AValue: TDateTime; const AYear, AMonth, ADay: Word): TDateTime; function RecodeXXXX(const AValue:	XXXX — Day, Hour, MilliSecond, Minute, Month, Second, Year. Заменяет нужные части указанной даты-времени

продолжение ➤

Подпрограмма	Описание
TDateTime; const AXXXX: Word): TDateTime;	
procedure ReplaceDate(var DateTime: TDateTime; const NewDate: TDateTime);	Заменяет дату
procedure ReplaceTime(var DateTime: TDateTime; const NewTime: TDateTime);	Заменяет время
function SameDate(const A, B: TDateTime): Boolean;	Возвращает True, если обе даты одинаковы
function SameTime(const A, B: TDateTime): Boolean;	Возвращает True, если оба времени одинаковы вплоть до миллисекунд
function SecondOf(const AValue: TDateTime): Word;	Возвращает количество секунд в указанном времени
function SecondOfTheXXXX(const AValue: TDateTime): LongWord;	XXXX — Day, Hour, Minute, Month, Week, Year. Возвращает количество секунд от начала указанного дня (часа, минуты, месяца, секунды, недели, года)
function SecondsBetween(const ANow, AThen: TDateTime): Int64;	Возвращает количество секунд между двумя датами
function StartOfAXXXX(): TDateTime;	XXXX — Day, Month, Week, Year. Возвращает первый момент дня (месяца, недели, года)
function Today: TDateTime;	Возвращает текущую дату и 00:00:00 время
function Tomorrow: TDateTime;	Возвращает завтрашнюю дату и 00:00:00 время
function WeekOf(const AValue: TDateTime): Word;	Возвращает неделю (1...53) указанной даты
function WeekOfTheXXXX(const AValue: TDateTime): Word;	XXXX — Month, Year. Возвращает порядковый номер недели в месяце (1...6) или в году (1...53)
function WeeksBetween(const ANow, AThen: TDateTime): Integer;	Возвращает количество недель между двумя датами
function WeeksInAYear(const AYear: Word): Word;	Возвращает количество полных недель в указанном году
function WithinPastXXXXs(const ANow, AThen: TDateTime; const AXXXXs: Integer): Boolean;	XXXX — Day, Hour, MilliSecond, Minute, Month, Second, Week, Year. Возвращает True, если обе даты отстоят не более чем на указанное количество дней (часов, миллисекунд, минут, месяцев, секунд, недель, лет)

Подпрограмма	Описание
function YearOf(const AValue: TDateTime): Word;	Возвращает год указанной даты
function YearsBetween(const ANow, AThen: TDateTime): Integer;	Возвращает количество лет между двумя датами
function Yesterday: TDateTime;	Возвращает вчерашнюю дату и 00:00:00 время

Спецификаторы формата даты/времени¹

Спецификатор	Описание
c	Отображает сначала дату в формате <i>дд.мм.гг</i> , затем пробел и время в формате <i>чч:мм:сс</i> : 08.06.02 19:45
d	Отображает день без ведущего нуля: 8
dd	Отображает день с ведущим нулем: 08
dddd	Отображает день недели: суббота (для нерусифицированной версии Windows — saturday)
dddddd	Отображает дату в формате <i>дд.мм.гг</i> : 08.06.02
ddddddd	Отображает дату в формате <i>д Месяц год</i> : 8 Июнь 2002 (для нерусифицированной версии Windows — 8 June 2002)
m	Отображает число месяца без ведущего нуля: 6
mm	Отображает число месяца с ведущим нулем: 06
mmm	Отображает сокращенное название месяца: июн
mmm	Отображает полное название месяца: Июнь
y или yy	Отображает две последние цифры года: 02
yyy или yyyy	Отображает все цифры года: 2002
h	Отображает час без ведущего нуля: 19
hh	Отображает час с ведущим нулем: 19
n	Отображает минуты без ведущего нуля: 45
nn	Отображает минуты с ведущим нулем: 45
s	Отображает секунды без ведущего нуля: 0
ss	Отображает секунды с ведущим нулем: 00
t	Отображает время в формате <i>чч:мм</i> : 19:45
tt	Отображает время в формате <i>чч:мм:сс</i> : 19:45:00
am/pm	Отображает время в 12-часовом формате (am — до полудня, pm — после полудня). Для спецификаторов hh:mm am/pm получим 07:45 pm

— продолжение ↗

¹ Примеры приводятся для даты 8 июня 2002 г. и времени 19:45:00.

Спецификатор	Описание
ampm	Отображает время в 12-часовом формате, но без указания до/после полудня. Для спецификаторов hh:mm ampm получим 07:45
a/p	Отображает время в 12-часовом формате (a — до полудня, p — после полудня). Для спецификаторов hh:mm a/p получим 07:45 p
/	Отображает используемый в Windows разделитель даты. Для спецификаторов d/m/y получим 8.6.02
:	Отображает используемый в Windows разделитель времени. Для спецификаторов h:n:s получим 19:45:0

Системные переменные, управляющие стандартным отображением данных

Переменная	Описание
CurrencyString: String;	Символ или символы денежной единицы; для русифицированной версии Windows ими являются символы «p.»
CurrencyFormat: Byte;	Определяет положение символов денежной единицы: 0 — \$1; 1 — 1\$; 2 — \$ 1; 3 — 1 \$; в русифицированной Windows используется формат 1 (1p.)
NegCurrFormat: Byte;	Определяет формат отрицательной суммы: 0 — (\$1); 1 — -\$1; 2 — \$-1; 3 — \$1-; 4 — (1\$); 5 — -1\$; 6 — 1-\$; 7 — 1\$-; 8 — -1 \$; 9 — -\$ 1; 10 — 1 \$-; 11 — \$ 1-; 12 — \$ -1; 13 — 1- \$; 14 — (\$ 1); 15 — (1 \$); в русифицированной Windows используется формат 5 (-1p.)
ThousandSeparator: Char;	Разделитель тысяч; в русифицированной Windows используется символ #166
DecimalSeparator: Char;	Разделитель дробной и целой частей числа, в русифицированной Windows используется запятая (',')
CurrencyDecimals: Byte;	Количество цифр после запятой в денежном формате. Обычно содержит 0, что блокирует вывод мелкой денежной единицы. Чтобы в сумме присутствовали не только рубли, но и копейки, в переменную следует установить значение 2
DateSeparator: Char;	Разделитель даты; в русифицированной Windows используется точка ('.')
ShortDateFormat: String;	Обычно используется формат 'dd.mm.yy', что соответствует, например, дате '08.06.02'
LongDateFormat: String;	Для русифицированной версии содержит символы 'dd MMMM yyyу г.', что дает '28 Июнь 2002 г.'

Переменная	Описание
TimeSeparator: Char;	Разделитель времени; в русифицированной Windows используется двоеточие (':')
ShortTimeFormat: String;	Краткий формат времени (обычно 'h:mm')
LongTimeFormat: String;	Полный формат времени (обычно 'hh:mm:ss')
ShortMonthNames: array [1..12] of String;	Краткие имена месяцев (янв, фев, мар, апр, май, июн, июл, авг, сен, окт, ноя, дек)
LongMonthNames: array [1..12] of String;	Полные имена месяцев (Январь, Февраль, Март, Апрель, Май, Июнь, Июль, Август, Сентябрь, Октябрь, Ноябрь, Декабрь)
ShortDayNames: array [1..7] of String;	Краткие имена дней недели (Вс, Пн, Вт, Ср, Чт, Пт, Сб)
LongDayNames: array [1..7] of String;	Полные имена дней недели (воскресенье, понедельник, вторник, среда, четверг, пятница, суббота)
TListSeparator: Char;	Разделитель списка строк (';')

Правила использования параметров функции FloatToStrF¹

Значение Format	Описание
ffExponent	Научная форма представления с множителем eXX («умножить на 10 в степени XX»). Precision задает общее количество десятичных цифр мантиисы, Digits — количество цифр в десятичном порядке XX. Число округляется с учетом первой отбрасываемой цифры: 3,1416E+00
ffFixed	Формат с фиксированным положением разделителя целой и дробной частей. Precision задает общее количество десятичных цифр в представлении числа, Digits — количество цифр в дробной части. Число округляется с учетом первой отбрасываемой цифры: 3,14
ffGeneral	Универсальный формат, использующий наиболее удобную для чтения форму представления вещественного числа. Соответствует формату ffFixed, если количество цифр в целой части меньше или равно Precision, а само число — больше или равно 0,00001, в противном случае соответствует формату ffExponent: 3,1416
ffNumber	Отличается от ffFixed использованием символа-разделителя тысяч при выводе больших чисел (для русифицированной версии Windows таким разделителем является пробел). Для Value = $\pi \times 1000$ получим 3 141,60
ffCurrency	Денежный формат. Соответствует ffNumber, но в конце строки ставится символ денежной единицы (для русифицированной версии Windows — символы «р.»). Для Value = $\pi \times 1000$ получим 3 141,60р.

¹ Примеры преобразования даются для Value = π (3,141593654), Precision = 5 и Digits = 2.

Спецификаторы форматирования вещественных чисел

Спецификатор	Описание
0	Определяет поле для цифры. Если в данной позиции форматируемое число имеет значащую цифру, она выводится, если нет — выводится 0
#	Определяет поле для цифры. Если в данной позиции форматируемое число имеет значащую цифру, она выводится, если нет — ничего не выводится
.	Поле для разделителя целой и дробной частей числа
,	Поле для разделителя тысяч
E+,e+,E- ,e-	Признак представления числа в научном формате. В этом случае число представляется мантиссой и десятичным порядком, между которыми стоит символ E. Спецификаторы E+ и e+ предписывают выводить знак + перед неотрицательным десятичным порядком, при использовании E- и e- знак + перед порядком не выводится
;	Разделитель спецификаторов формата для положительного, отрицательного и нулевого числа

Подпрограммы для работы с файлами

Подпрограмма	Описание
function ChangeFileExt (const FileName, Extension: String): String ;	Изменяет существующее расширение файла расширением, заданным параметром Extension
Procedure ChDir(Path: String);	Изменяет текущий каталог: Path — строковое выражение, содержащее путь к устанавливаемому по умолчанию каталогу
function DateTimeToFileDate (DateTime: TDateTime): Integer;	Преобразует значение DateTime в системный формат времени создания (обновления) файла
Function DiskFree(D: Byte): LongInt;	Возвращает объем в байтах свободного пространства на указанном диске: D — номер диска (0 — устройство по умолчанию, 1 — диск A, 2 — диск B и т. д.). Функция возвращает значение -1, если указан номер несуществующего диска
Function DiskSize(D: Byte): LongInt;	Возвращает объем в байтах полного пространства на указанном диске: D — номер диска (0 — устройство по умолчанию, 1 — диск A, 2 — диск B и т. д.). Функция возвращает значение -1, если указан номер несуществующего диска

Подпрограмма	Описание
function FileAge (const FileName: String): Integer;	Для файла FileName возвращает время его последнего обновления (в системном формате) или -1, если такого файла не существует
function ExcludeTrailingBackslash (const S: String): String;	Исключает из строки S замыкающий символ «\» (если этот символ не замыкает строку, возвращает S без изменения)
function ExpandFileName (const FileName: String): String;	Дополняет имя файла текущим каталогом (и диском)
function ExpandUNCFileName (const FileName: String): String;	Дополняет имя файла текущим сетевым каталогом (и диском)
function ExtractFileDir (const FileName: String): String;	Извлекает из полного имени файла маршрут доступа к нему (без последнего символа «\»)
function ExtractFileExt (const FileName: String): String;	Извлекает из полного имени файла его расширение (с ведущей точкой)
function ExtractFileName (const FileName: String): String;	Извлекает из полного имени файла его имя (с расширением)
function ExtractFilePath (const FileName: String): String;	Извлекает из полного имени файла маршрут доступа к нему (с последним символом «\»)
function ExtractRelativePath (const BaseName, DestName: String): String;	Извлекает из полного имени файла имя маршрута относительно DestName (промежуточные каталоги заменяются символами «..\»)
function ExtractShortPathName (const FileName: String): String;	Преобразует имя файла к короткому формату 8.3 для MS-DOS и Windows 3.x
function FileDateToDateTime (FileDate: Integer): TDateTime;	Преобразует системный формат FileDate времени создания файла в формат дата-время
function FileGetDate (Handle: Integer): Integer;	По заданному дескриптору файла Handle возвращает время и дату его создания в системном формате. Возвращает 0 в случае успеха или код ошибки
function FileSetDate (Handle: Integer; Age: Integer): Integer;	Для файла с дескриптором Handle устанавливает новое время и дату его создания Age в системном формате. Возвращает 0 в случае успеха или код ошибки
function IncludeTrailingBackslash (const S: String): String;	Возвращает полный маршрут доступа к файлу с ведомым символом «\»
Function IOResult: Integer;	Возвращает условный признак последней операции ввода-вывода
function IsPathDelimiter (const S: String; Index: Integer): Boolean;	Возвращает True, если в строке S символ Index есть «\»

продолжение ➤

Подпрограмма	Описание
function MatchesMask (const FileName, Mask: String): Boolean;	Возвращает True, если имя FileName соответствует групповому имени Mask
procedure ProcessPath (const EditText: String ; var Drive: Char; var DirPart: String ; var FilePart: String);	Возвращает имя диска, маршрут поиска и имя файла в переменных Drive, DirPart и FilePart соответственно; EditText — полное имя файла
Procedure Rmdir (Dir: String);	Удаляет каталог Dir. Удаляемый каталог должен быть пустым, то есть не содержать файлов или имен каталогов нижнего уровня

Стандартные классы исключений

Класс	Родитель	Обрабатываемое исключение
EAbort	Exception	Реализует «тихую» (без какого-либо сообщения) обработку любого исключения
EAbstractError	Exception	Программа пытается вызвать абстрактный метод
EAccessViolation	Exception	Программа пыталась обратиться к не принадлежащей ей области памяти или использует недействительный указатель
EAppletException	Exception	Ошибка связана с созданием управляющих панелей в апплетах
EArrayError	Exception	Возникает из-за различного рода ошибок при работе с массивами (неверный индекс, попытка вставить элемент в массив фиксированной длины или в отсортированный массив и т. п.)
EAssertionFaild	Exception	Возбуждается отладочной процедурой Assert, когда тестируемое ею логическое выражение имеет значение False
EBitsError	Exception	Программа пыталась обратиться к свойству Bits объекта TBits с индексом меньше нуля или больше максимально допустимого значения
EBrokerException	Exception	Объект-брокер не может найти сервер
ECacheError	Exception	Ошибка в наборе данных для компонента TDecisionCube
EClassNotFound	EFilerError	Для компонента, читаемого из потока данных, не найден соответствующий класс. Обычно возникает в случае, когда в форму вставлен нестандартный

Класс	Родитель	Обрабатываемое исключение
		компонент, а в библиотеке компонентов Delphi нет связанного с ним класса
ECommonCalendarError	Exception	Возникает в объектах класса TCommonCalendar и его потомках, когда вводится неверная дата
EComponentError	Exception	Возникает при различных манипуляциях программы с компонентом (программа не может зарегистрировать компонент, переименовать его или для его работы требуется интерфейс COM, который компонентом не поддерживается)
EControlC	Exception	Возникает при нажатии клавиш Ctrl+C в случае работы приложения в режиме консоли
EConvertError	Exception	Ошибка преобразования в функциях StrToInt или StrToFloat
ECorbaDispatch	Exception	Возникает в программах, использующих технологию CORBA, при ошибках, связанных с несовпадением интерфейсов сервера и брокера данных
ECorbaException	Exception	Возникает в программах, использующих технологию CORBA
ECorbaUserException	ECorba-Exception	Возникает как определяемая пользователем реакция на ошибки интерфейса
EDatabaseError	Exception	Возникает, когда компонент обнаруживает ошибку в базе данных
EDatetimeError	Exception	Возбуждается компонентом TDateTimePicker при попытке ввода неверной даты или времени
EDBClient	EDatabase-Error	Ошибка связана с неправильной работой TClientDataSet
EDBEditError	Exception	Возникает, когда компонент пытается использовать данные, несовместимые с заданной маской
EDBEngineError	EDatabase-Error	Связана с ошибками BDE
EDimensionMarError	Exception	Возникает, когда используемый в кубе решений набор данных не имеет агрегатных полей
EDimIndexError	Exception	Связана с нарушением размерности массива данных для куба решений
EDivByZero	EIntError	Ошибка целочисленного деления на ноль

— продолжение ➤

Класс	Родитель	Обрабатываемое исключение
EDSWriter	Exception	Ошибка при подготовке провайдером пакета данных для набора данных
EExternalException	EStreamError	Возникла ошибка, код которой не является предопределенным в Delphi
ECreateError	EStreamError	Ошибка при создании файла. Например, попытка создать файл на устройстве, предназначенном только для чтения, или в несуществующем каталоге
EFileError	EStreamError	Программа пытается повторно зарегистрировать в потоке данных один и тот же класс
EOpenError	EStreamError	Ошибка открытия потока данных. Например, попытка открыть несуществующий файл
EHeapException	Exception	Ошибка связана с неправильными операциями с динамической памятью
EIBClientError	EIBError	Ошибка связана с функционированием IBX-клиента
EIBError	EDatabaseError	Общая ошибка технологии IBX
EIBInterbaseError	EIBError	Ошибка связана с функционированием сервера в технологии IBX
EInOutError	Exception	Любая ошибка в файловых операциях. Поле ErrorCode объекта этого класса содержит код ошибки
EInterpreterError	Exception	Возникает, когда компонент класса TDataBlockInterpreter не может интерпретировать данные блока данных
EIntError	Exception	Любая ошибка в целочисленных вычислениях
EIntfCastError	Exception	Попытка недопустимого приведения типов в OLE-объектах
EIntOverflow	EIntError	Ошибка целочисленного переполнения: программа пытается присвоить целочисленной переменной значение, выходящее из 32 двоичных разрядов
EInvalidArgument	EMatchError	Возбуждается математическими функциями при выходе аргумента из допустимого диапазона
EInvalidCast	Exception	Программа пытается осуществить недопустимое преобразование типов с помощью оператора as
EInvalidGraphic	Exception	Программа пытается загрузить в контейнер изображение из файла,

Класс	Родитель	Обрабатываемое исключение
		который имеет недопустимый формат (допустимыми форматами являются растр, метафайл, курсор, значок)
EInvalidGraphic-Operation	Exception	Программа пытается выполнить недопустимую графическую операцию
EInvalidGridOperation	Exception	Программа пытается выполнить недопустимую операцию с таблицей (например, обратиться к несуществующим столбцу или строке)
EInvalidImage	EFileError	Программа пытается прочитать ресурс изображения из файла, в котором этого ресурса нет
EInvalidOp	EMatchError	Ошибка в операциях с плавающей точкой (недопустимая инструкция, переполнение стека сопроцессора и т. п.)
EInvalidOperation	Exception	Не имеющий окна компонент пытается выполнить операцию, требующую дескриптора окна
EInvalidPointer	EHeap-Exception	Попытка использовать недействительный указатель
EListError	Exception	Эта ошибка связана с неверными действиями программы по отношению к разного рода спискам. Например, обращение к элементу списка с индексом меньше нуля или больше максимально допустимого
ELowCapacityError	Exception	Ошибка возникает при попытке выделения памяти на устройстве, у которого нет нужной свободной памяти
EMatchError	Exception	Любая ошибка при выполнении вычислений с плавающей точкой
EMenuError	Exception	Ошибка при работе программы с меню. Например, при добавлении элемента с идентификатором, который уже определен в меню
EMCIDiviceError	Exception	Ошибка возникла в медиаплеере
EMethodNotFound	EFileError	Программа прочитала из потока данных объект, но не может найти связанный с классом объекта метод
EMonthCalError	ECommonCalendarError	Возбуждается компонентом класса TMonthCalendar при попытке ввода неправильной даты
EOleCtrlError	Exception	Программа не может установить связь с элементом ActiveX

— продолжение ➤

Класс	Родитель	Обрабатываемое исключение
EOleError	Exception	Низкоуровневая ошибка в технологии OLE
EOleException	EOleSysError	Программа использует неверный OLE-интерфейс
EOleRegistrationError	EOleError	Ошибка регистрации OLE-объекта в реестре Windows
EOleSysError	EOleError	Возникает при неправильном выполнении команды OLE автоматизации
EOutlineError*	Exception	Возникает при ошибке доступа к компоненту класса TOutLine
EOutOfMemory	EHeap-Exception	Эта ошибка возникает, когда программа запрашивает слишком большой для данной конфигурации Windows объем памяти
EOutOfResource	EOutOfMemory	Программа требует от Windows дескриптор окна, но операционная система исчерпала лимит дескрипторов
EOverflow	EMatchError	Результат операций с плавающей точкой слишком велик, чтобы уместиться в регистрах сопроцессора
EPackageError	Exception	Возникает при ошибке доступа к пакету
EParserError	Exception	Ошибка преобразования текста в двоичные данные при чтении из потока
EPrinter	Exception	Система Windows сообщила программе об ошибке принтера
EPrivilege	Exception	Программа пытается выполнить привилегированную операцию. Привилегированные операции могут выполняться только ядром Windows
EPropertyConvertError	Exception	Ошибка при чтении или записи значения свойства
EPropertyError	Exception	Ошибка доступа к свойству при чтении или записи
EPropReadOnly	Exception	Программа пытается присвоить значение свойству, из которого можно только читать (при использовании технологии OLE)
EPropWriteOnly	Exception	Программа пытается прочитать свойство, предназначенное только для записи
ERangeError	EIntError	Целочисленный результат превышает емкость целого типа данных

Класс	Родитель	Обрабатываемое исключение
EReadError	EFileError	Программа не может прочитать из потока данных нужного количества байтов
EReconcileError	EDatabaseError	Ошибка обновления данных в TClientDataset
ERegistryException	Exception	Ошибка, связанная с операцией над реестром Windows
EResNotFound	Exception	Программа не может найти указанный ресурс в файле ресурсов
ESocketConnectionError	Exception	Ошибка связана с работой с сокетами Windows
ESocketError	Exception	Ошибка связана с работой с сокетами Windows
EStackOverflow	Exception	Исчерпан объем выделенного программе стека
EStreamError	Exception	Любая ошибка при работе с потоком данных
EStringListError	Exception	Программа ссылается на строку, индекс которой выходит из диапазона возможных значений для списка строк
EThread	Exception	Ситуация борьбы за общий ресурс в программе с несколькими потоками команд
ETreeViewError к TTreeView	Exception	Указан неверный индекс при обращении
EUnderflow	EMatchError	Результат операций с плавающей точкой слишком мал, чтобы уместиться в регистрах сопроцессора (исчезновение порядка)
EUnsupportedTypeError	Exception	Выбран недопустимый тип поля в качестве измерения в кубе решений
EUpdateError	Exception	Ошибка обновления набора данных провайдера
EVariantError	Exception	Ошибка при работе с типом Variant: недопустимое приведение типов; недопустимая операция; обращение к скалярной переменной как к варианту-массиву; индекс варианта-массива выходит из допустимых значений
EWin32Error	Exception	Ошибочное обращение к API-функции Windows. Свойство Message содержит номер ошибки и связанное с ней сообщение
EWriteError	EFileError	Ошибка записи в поток данных
EZeroDivide	EMatchError	Вещественное деление на ноль

Литература

1. Архангельский А. Я. Delphi 6. Справочное пособие. — М.: ЗАО «Издательство БИНОМ», 2001.
2. Бобровский С. Delphi 6 и Kylix: библиотека программиста. — СПб.: Питер, 2002.
3. Дарахвелидзе П. Г., Марков Е. П. Программирование в Delphi 7. — СПб.: БХВ-Петербург, 2003.
4. Конопка Р. Создание оригинальных компонент в среде Delphi/Пер. с англ. Рэй Конопка. — К.: НИПФ «ДиаСофт Лтд.», 1996.
5. Культин Н. Б. Основы программирования в Delphi 7. — СПб.: БХВ-Петербург, 2003.
6. Марко Канту, Тим Гуч при участии Джона Лэма. Delphi. Руководство разработчика: Пер. с англ. — К.: ВЕК+, М.: ЭНТРОП, М.: ДЭСС, 1999.
7. Тейксейра Стив, Пачек Ксавье. Delphi 6. Руководство разработчика/Пер. с англ. — М.: Издательский дом «Вильямс», 2002.

Алфавитный указатель

А

адрес, 146
анимация, 362

Б

библиотека типов, 529
буква, 103

В

вариант, 184
выравнивания компонентов, 572
значения, 184
преобразования, 186
работа, 187
структура, 185
тип данных, 523
вешка
перемещения, 392
разбивки, 339
всплывающая подсказка, 288
выражение, 79, 106

Г

главное меню формы, 296
главное окно, 27
глобально уникальный
идентификатор, 517
группа
переключателей, 312
флажков, 338

Д

дата-время, 366
действие, 283, 298, 314
деструктор, 176
диаграмма, 349
диалоговое окно
выбора
цвета, 426
шрифта, 424
настройки параметров
печати, 426
принтера, 428

диалоговое окно (*продолжение*)

открытия
изображений, 423
файлов, 420
поиска, 428
поиска и замены, 430
сохранения
изображений, 423
файлов, 420

динамически подключаемая
библиотека, 503

директива

abstract, 175
assembler, 156
dynamic, 174
external, 156
far, 156
Forward, 169
forward, 156
interrupt, 156
near, 156
override, 174
virtual, 174

директивы

компилятора, 62
стандартные, 104, 156

Е

единица программная, 61

З

заглушка, 96, 527
заголовков
подпрограммы, 98
программы, 62
процедуры, 96
управляющий, 383
заместитель, 527
запись, 129
поле вариантное, 130
разделы, 129
зарезервированное слово, 62, 67, 103
and, 79

зарезервированное слово *(продолжение)*

array, 92, 125
as, 521
automated, 180
begin, 63
case, 87, 130
class, 177
const, 67, 99
constructor, 176
default, 550
destructor, 176
dispid, 522
dispinterface, 522
div, 73
do, 82
downto, 84
else, 78
end, 61, 215
exports, 504
file, 192
finalization, 215
finally, 230
for, 82
goto, 90
if, 78
implementation, 64
implements, 521
in, 133, 455
Inherited, 177
initialization, 215
interface, 64, 215, 516
label, 68, 91, 99
library, 504
message, 498
mod, 73
nill, 149
not, 79
of, 87, 92, 125, 130, 133, 192
onerror, 177
or, 79
out, 161
packed, 125
private, 180
procedure, 96
program, 61, 455
property, 179
protected, 180
public, 180
published, 180
raise, 233
read, 179
reintroduce, 178
repeat, 86
set, 133
string, 136

зарезервированное слово *(продолжение)*

then, 78
to, 82
try, 230
type, 66
unit, 64, 215
until, 86
uses, 62, 215, 455
var, 66, 99
with, 130
write, 179
защищенный блок, 83, 230
звук, 405, 412
значок, 257

И

идентификатор, 67, 104
 глобально уникальный, 517
 интерфейса, 517
иерархический список, 378
иерархическое дерево, 369
изображение, 334
имя
 глобальное, 154
 локальное, 153
индекс массива, 92
индикатор
 величины, 432
 процесса, 360
инкапсуляция, 171
интерфейс, 526
 Drag&Dock, 286
 Drag&Drop, 283
 IClassFactory, 528
 IDispatch, 522
 IUnknow, 526
интерфейсы, 516
 делегирование, 521
 дуальные, 524
исключение, 84, 229
 возбуждение, 233
 обработчик, 230
 стандартные классы, 232
исключительная ситуация, 229

К

календарь, 368
канва, 253
кисть, 250
клавиши быстрого вызова, 280, 296, 298
класс, 66
 EAbort 622
 EAbstractError 622
 EAccessViolation 622
 EAppletException 622

класс (*продолжение*)

EArrayError 622
 EAssertionFailed 622
 EBitsError 622
 EBrokerException 622
 ECasheError 622
 EClassNotFound 622
 ECommonCalendarError 623
 EComponentError 623
 EControl 623
 EConvertError 623
 ECorbaDispatch 623
 ECorbaException 623
 ECorbaUserException 623
 EDatabaseError 623
 EDateTimeError 623
 EDBClient 623
 EDBEditError 623
 EDBEngineError 623
 EDimensionMapError 623
 EDimIndexError 623
 EDivByZero 623
 EDSWriter 624
 EExternalException 624
 EFCREATEError 624
 EFFilterError 624
 EFOpenError 624
 EHeapException 624
 EIBClientError 624
 EIBError 624
 EIBInterbaseError 624
 EInOutError 624
 EInterpreterError 624
 EIntError 624
 EIntfCastError 624
 EIntOverflow 624
 EInvalidArgument 624
 EInvalidCast 624
 EInvalidGraphic 624
 EInvalidGraphicOperation 625
 EInvalidGridOperation 625
 EInvalidImage 625
 EInvalidOp 625
 EInvalidOperation 625
 EInvalidPointer 625
 EListError 625
 ELowCapacityError 625
 EMatchError 625
 EMCIDeviceError 625
 EMenuError 625
 EMethodNotFound 625
 EMonthCalError 625
 EOleCtrlError 625
 EOleError 626
 EOleException 626

класс (*продолжение*)

EOleRegistrationError 626
 EOleSysError 626
 EOutlineError 626
 EOutOfMemory 626
 EOutOfResource 626
 EOverflow 626
 EPackageError 626
 EParserError 626
 EPrinter 626
 EPrivilege 626
 EPropertyConverError 626
 EPropertyError 626
 EPropReadOnly 626
 EPropWriteOnly 626
 ERangeError 626
 EReadError 627
 EReconcileError 627
 ERegistryException 627
 EResNotFound 627
 ESocketConnectionError 627
 ESocketError 627
 EStackOverflow 627
 EStreamError 627
 EStringListError 627
 EThread 627
 EtreeViewError 627
 EUnderflow 627
 EUnsupportedTypeError 627
 EUpdateError 627
 EVariantError 627
 EWin32Error 627
 EWriteError 627
 Exception, 229, 231
 EZeroDivide 627
 TAction, 315
 TBrush, 250
 TCanvas, 253
 TComponent, 261, 544
 TControl, 264, 544
 TCustomControl, 544
 TFont, 243
 TForm, 66
 TGraphicControl, 261, 544
 TGraphic, 257
 THeaderSection, 385
 TIconOption, 382
 TInterfacedObject, 519
 TList, 235
 TListItems, 382
 TMonthCalendar, 368
 TObject, 66, 543
 TParaAttributes, 357
 TPen, 246

класс (*продолжение*)

TPersistent, 261, 544
 TPicture, 258
 TRegistry, 480
 TStatusPanel, 387
 TStatusPanels, 387
 TStream, 211
 TStringList, 240
 TStrings, 239
 TToolButton, 389
 TTreeNode, 374
 TWinControl, 261, 544

классы, 170

именование, 44
 инкапсуляция, 171
 компонентные, 518
 методы, 173
 наследование, 171
 определение, 44
 полиморфизм, 172
 поля, 77, 172
 свойства, 179
 секции объявления, 180
 стандартные, 44
 экземпляры, 44, 45

кнопка

обычная, 304
 панели инструментов, 319, 389, 390
 с изображением, 317, 319
 с фиксацией, 319
 спаренная, 361

код

ANSI, 115
 виртуальный, 279, 281

коллекция, 539

комбинированный список, 310

комментарий, 62, 100

компонент, 30, 48

TActionList, 314
 TActionManager, 350
 TAnimate, 362
 TBevel, 337
 TBitBtn, 49, 317
 TButton, 304
 TCalendar, 436
 TChart, 349
 TChartFX, 54
 TCheckBox, 49, 305
 TCheckBoxList, 50, 338
 TColorDialog, 426
 TColorGrid, 433

компонент (*продолжение*)

TComboBox, 310
 TControlBar, 341
 TCoolBand, 392
 TDateTimePicker, 366
 TDirectoryOutLine, 435
 TDrawGrid, 330
 TEdit, 49, 299
 TFindDialog, 428
 TFontDialog, 424
 TForm, 439, 442
 TGauge, 432
 TGroupBox, 312
 THeaderControl, 383
 TImage, 334
 TImageList, 356
 TLabel, 49, 298
 TListBox, 306
 TListView, 378
 TMainMenu, 296
 TMaskEdit, 320
 TMediaPlayer, 404
 TMemo, 302
 TOleContainer, 413
 TOpenDialog, 420
 TOpenPictureDialog, 423
 TPageControl, 355
 TPageScroller, 394
 TPaintBox, 403
 TPanel, 49, 313
 TPopupMenu, 298
 TPrintDialog, 426
 TPrinterSetupDialog, 428
 TProgressBar, 51, 360
 TRadioButton, 49, 306
 TRadioGroup, 312
 TReplaceDialog, 430
 TRichEdit, 356
 TSaveDialog, 420
 TSavePictureDialog, 423
 TScrollBar, 311
 TScrollBox, 50, 337
 TShape, 50, 336
 TShellChangeNotifier, 437
 TShellComboBox, 437
 TShellListView, 437
 TShellTreeView, 437
 TSpeedButton, 319
 TSpinButton, 434
 TSpinEdit, 434
 TSplitter, 50, 339
 TStaticText, 50, 340

компонент (*продолжение*)

TStatusBar, 387
 TStringGrid, 50, 321
 TTabControl, 353
 TTimer, 400
 TToolBar, 389
 TTrackBar, 360
 TTreeView, 369, 370
 TUpDown, 361
 TValueListEditor, 344

компоненты

владелец, 263
 дочерние, 264
 клиентская часть, 267
 неоконные, 262
 оконные, 262
 определение, 261
 родительские, 264
 константа, 67, 105
 логическая, 105
 символьная, 105
 строковая, 106
 типизированная, 99, 222
 конструктор, 176
 массива, 164
 меню, 296
 множества, 106
 контейнер, 58
 изображений, 356
 панели инструментов, 341
 с вкладками, 353
 контекстное меню, 298
 конфигурация окон Delphi, 30
 кромка, 337
 куча, 146, 147

Л

литерал, 320

М

маска ввода, 320
 массив, 91, 125
 вариантный, 188
 динамический, 127
 индекс, 92
 конструктор, 164
 открытый, 163
 менеджер действий, 350
 меню
 главное, 296
 контекстное, 298
 многоуровневое, 297
 со значками, 298
 метафайл, 258
 метка, 68, 90, 298, 340

метод

Application.Initialize, 63
 BeginDrag, 286
 BringToFront, 268
 DisableAlign, 268
 DragDrop, 286
 EnabledAlign, 268
 EndDrag, 286
 Free, 176
 GetTextBuf, 269
 GetTextLen, 269
 GetTabOrderList, 283
 Hide, 268
 SendToBack, 268
 SetFocus, 282
 SetTextBuf, 269
 Show, 268
 TApplication.CreateForm, 63
 TApplication.Run, 64
 TStringList.Add, 71
 TThread.Synchronize, 491

методы

абстрактные, 175
 виртуальные, 172, 174
 деструктор, 176
 динамические, 174
 класса, 173
 конструктор, 176
 одноименные, 177
 перекрытие, 172

многострочное текстовое поле, 302

множество, 132

включение в другое, 132
 конструктор, 133
 операции, 133
 пустое, 132
 эквивалентность другому, 132

модуль, 40, 64, 214

данных, 221
 динамической библиотеки, 221, 503
 завершающая часть, 217
 заголовок, 215
 инициализирующая часть, 217
 интерфейсная часть, 216
 интерфейсный, 510
 исполняемая часть, 216
 потока, 221, 489
 терминатор, 64
 форма, 220

Н

набор

вкладок, 353
 страниц с вкладками, 355
 наследование, 171

О

обработчик
 событий, 44
 сообщений Windows, 343
 объект
 Application, 63, 460
 Clipboard, 473
 OLE, 535
 Printer, 468
 TObject, 171
 TScreen, 466
 COM, 526
 объектно-ориентированное
 программирование, 63
 объекты
 автоматизации, 522
 внедренные, 413
 методы, 63
 определение, 63, 170
 отображения, 205
 поля, 63
 связанные, 413
 однострочное текстовое поле, 299
 окно
 браузера, 25
 всплывающее, 440
 выбора шаблона, 295
 главное, 25, 27
 дескриптор, 262
 диалоговое, 419
 для рисования, 403
 дочернее, 439
 инспектора объектов, 25, 32
 кода, 25, 34
 модальное, 304, 452
 немодальное, 452
 рамочное, 439
 регистрации шаблона, 294
 ресурс, 261
 сервера, 414
 формы, 25, 31
 оперативная подсказка, 288
 оператор
 выбора, 86
 перехода, 90
 присваивания, 46
 присоединения, 130
 пустой, 78
 составной, 77
 условный, 78
 цикла
 for, 82
 repeat...until, 86
 while, 84

операция, 107
 логическая, 79, 108
 математическая, 73
 отношения, 70
 приоритет, 107
 сдвиговая, 109
 сцепления, 70
 открытый массив, 163

П

пакет, 221
 палитра компонентов, 30
 память
 адрес, 146
 выделение, 147
 динамическая, 146
 оперативная, 146
 освобождение, 148
 отображаемая, 494
 панель
 без заголовка, 313
 группирования, 312
 инструментов, 341, 389, 392
 с кнопками прокрутки, 394
 с полосами прокрутки, 337
 строки состояния, 387
 параметр
 вариантный массив, 165
 значение, 159
 именованный, 523
 константа, 159
 нетипизированный, 161
 переменная, 159
 позиционный, 523
 умалчиваемый, 162
 пароль, 321
 переключатель, 306, 312
 переменная
 result, 98
 глобальная, 160
 инициализация, 99
 инициализированная, 222
 локальная, 537
 определение, 68
 файлового типа, 193
 перо, 246
 пиксел, 42
 подпрограмма, 68, 94, 152
 вызов, 152
 опережающее описание, 169
 описание, 152
 параметр
 вариантный массив, 165
 вызова, 45
 значение, 160

подпрограмма, параметр (*продолжение*)
 константа, 160
 нетипизированный, 161
 открытый массив, 163
 переменная, 160
 умалчиваемый, 162
 рекурсия, 154
 стандартная, 95
 формальные параметры, 157
 поле
 многострочное текстовое, 302
 однострочное с меткой, 348
 однострочное текстовое, 299
 с маской ввода, 320
 формата RTF, 356
 ползунок, 360
 полиморфизм, 172
 полоса
 панели инструментов, 392
 прокрутки, 311
 поток
 данных, 211
 команд, 221, 486, 489
 предложение языка программирования, 45
 признак конца
 строки, 199
 файла, 199
 приоритет операций, 107
 пробел, 103
 программа
 Average, 92
 Calc, 87
 CopyTxt, 71
 Epsilon, 85
 IntMult, 73
 IntSumm, 83
 IntTest, 80
 PrimSet, 134
 UpString, 95
 программирование нисходящее, 96
 программная единица
 определение, 61
 терминатор, 62
 программы
 визуализация загрузки, 456
 заголовков, 62
 защита паролем, 457
 исполнение, 39
 кодирование, 98
 компиляция, 39
 компоновка, 39
 консольные, 457
 учебные, 55
 элементы, 66
 произвольная таблица, 330

процедура, 94
 Append, 194
 AssignFile, 193, 194
 Beep, 412
 BlockRead, 204
 BlockWrite, 204
 break, 86
 ChDir, 620
 CloseFile, 194
 continue, 86
 DateTimeToString, 142
 Delete, 140
 Dispose, 148, 149
 Erase, 195
 Exclude, 133
 FilePos, 203
 Finalize, 127
 FindClose, 195
 Flush, 195
 FreeMem, 149, 150
 Frexp, 609
 GetDir, 195
 GetMem, 149, 150
 Include, 133
 Insert, 140
 MessageBeep, 82
 Mkdir, 195
 MomentSkewKurtosis, 610
 New, 147, 150
 randomize, 121
 Read, 200, 201, 203
 ReadLn, 200, 202
 Rename, 196
 Reset, 193, 196
 Rewrite, 193, 196
 Rmdir, 196, 622
 Seek, 203
 SetLength, 141
 Str, 143
 StrPCopy, 138
 Val, 142
 VarCast, 188
 VarClear, 188
 VarCopy, 188
 Write, 200, 202, 203
 WriteLn, 200, 203
 процедуры
 вызов, 94
 заголовков, 45, 96, 156
 оконные, 264, 495
 тело, 45, 96
 процесс, 486

Р

раскрывающийся список, 310
 растр, 258

редактор

- библиотеки типов, 531
 - действий, 314
 - иерархического дерева, 370
 - изображений, 341
 - меню, 296
 - списков специализированный, 344
- реестр Windows, 480
- рекурсия, 154, 167, 168

С

свойство, 179

- Action, 283
- Align, 267
- Alignment, 269
- Anchors, 274
- AutoSize, 274, 287
- BiDiMode, 292
- BoundsRect, 267
- CanFocus, 282
- ClientToScreen, 267
- Color, 270
- ComObject, 290
- ComponentCount, 263
- Components, 263
- Constraints, 274, 287
- ControlCount, 265
- Controls, 265
- ControlState, 291
- ControlStyle, 291
- Ctl3d, 268
- DesktopFont, 270
- DockOrientation, 287
- DragCursor, 284
- DragKind, 286
- DragMode, 284
- Enabled, 268
- FloatingDockSizeClass, 287
- Focused, 282
- Font, 269
- Height, 267
- HelpContext, 290
- Hint, 288
- HintColor, 290
- HintHidePause, 290
- HintPause, 290
- HintShortPause, 290
- ImeMode, 292
- ImeName, 292
- KeyPreview, 279
- Left, 267
- LRLockWidth, 287
- Name, 58
- Owner, 263
- Parent, 264

свойство (*продолжение*)

- .ParentColor, 273
 - ParentCtl3d, 273
 - ParentFont, 273
 - ParentHint, 273
 - PopupMenu, 290
 - ScreenToClient, 267
 - ShowHint, 288
 - TabOrder, 283
 - TabStop, 283
 - Tag, 82, 263
 - TBDockHeight, 287
 - TBitBtn.Kind, 59
 - TEdit.Left, 58
 - TEdit.Text, 59
 - TEdit.Top, 58
 - TEdit.Width, 58
 - TForm.Align, 58
 - TForm.Height, 57
 - TForm.Position, 58
 - TForm.Width, 57
 - TMaskEdit.EditMask, 75
 - TMemo.Align, 59
 - TMemo.Lines, 59
 - TMemo.ScrollBars, 59
 - TMemo.WordWrap, 59
 - Top, 267
 - TPanel.BevelOuter, 58
 - TPanel.Caption, 58
 - UnlockHeight, 287
 - UnlockWidth, 287
 - VCLComObject, 290
 - Visible, 268
 - Width, 267
- секретность ввода, 321
- сервер
- OLE, 413
 - регистрация, 528
 - служба, 526
- серия, 349
- символ
- секретный, 321
 - специальный, 103
- событие
- OnClick, 278
 - OnCreate, 46
 - OnDblClick, 278
 - OnDockDrop, 287
 - OnDockOver, 287
 - OnDragDrop, 285
 - OnDragEvent, 284
 - OnEndDrag, 285
 - OnGetSiteInfo, 288
 - OnIdle, 466

событие (*продолжение*)

- OnKeyDown, 279
- OnKeyPress, 279
- OnKeyUp, 279
- OnMouseDown, 278
- OnMouseMove, 278
- OnMouseUp, 278
- OnUnDock, 288

- сообщение, 343, 495

- список, 235

- выбора цвета, 348
 - действий, 314
 - иерархический, 378
 - комбинированный, 310
 - обычный, 306
 - раскрывающийся, 310
 - с расширенными возможностями, 395
 - сортировка строк, 241
 - строк и объектов, 239

- стандартная фигура, 336

- стек, 168

- строка

- заканчивающаяся нулем, 136
 - короткая, 136
 - состояния, 387
 - форматирующая, 144
 - широкая, 136

- счетчик, 361

T

- таблица

- произвольная, 330
 - текстовая, 321

- таймер, 400

- текст

- ввод и отображение, 302
 - формата RTF, 356

- текстовая метка, 340

- текстовая таблица, 321

- тело процедуры, 96

- терминатор программной единицы, 62

- технология

- DDE, 52, 525
 - OLE, 52, 525
 - COM, 525

- тик, 400

- тип, 67, 69

- Char, 69
 - String, 69, 136
 - TFindOptions, 428
 - TFontDialogOption, 425
 - TGridOptions, 324
 - TMPTimeFormats, 409
 - TOpenOptions, 421

- типы

- варианты, 184
 - вещественные, 85, 111, 119
 - дата-время, 111, 123
 - диапазоны, 92, 118
 - записи, 129
 - индексные, 125
 - классы, 170
 - логические, 71, 114
 - массивы, 125
 - множества, 132
 - определение, 110
 - перечисленные, 116
 - порядковые, 111
 - приведение, 250
 - простые, 111
 - процедурные, 166
 - псевдонимы, 150
 - символьные, 114
 - строковые, 136
 - структурированные, 125
 - указатели, 146
 - файлы, 192
 - целые, 73, 112

У

- указатель, 146

- мышь

- нестандартный, 275
 - стандартный, 275

- нетипизированный, 147
 - разыменование, 148

- типизированный, 146

- управляющий заголовок, 383

Ф

- фабрика классов, 528

- файл, 192

- динамической подкачки, 205

- инициализации, 475

- инициализация, 193

- модуля, 458

- настройки, 459

- нетипизированный, 203

- отображение в память, 205

- переменная, 193

- подкачки, 494

- проекта, 61, 454

- резервный, 460

- ресурсов, 277, 459

- текстовый, 199

- типизированный, 203

- флажок, 305

- фокус ввода, 72, 282

- форма, 439

форматирующий спецификатор, 144

фрейм, 293

функции

вызов, 152

заголовок, 156

преобразования типа, 117

функция, 68, 95

abs, 112, 121

Addr, 149

AnsiLowerCase, 140

AnsiUpperCase, 140

ArcCos, 607

ArcCosH, 608

ArcSin, 607

ArcSinH, 608

arctan, 121

ArcTan2, 607

ArcTanH, 608

Ceil, 609

char, 97

chr, 112, 116

CloseHandle, 207

Concat, 140

Copy, 140

cos, 121

CosH, 608

Cotan, 607

CreateFileMapping, 205

CreateOleObject, 535

CreateProcess, 486

CycleToRad, 608

Date, 123

DateTimeToFile, 620

DateTimeToStr, 123, 142

DateToStr, 123, 142

dec, 112

DegToRad, 608

DiskFree, 620

DiskSpace, 620

DoubleDecliningBalance, 611

EOF, 195

Eoln, 200

exp, 121

FileAge, 621

FileCreate, 205

FileDateTimeToFile, 621

FileExists, 194, 195, 421

FileGetDate, 621

FileOpen, 205

FileSetDate, 621

FileSize, 203

FindFirst, 195

FindNext, 195

FloatToStr, 86, 142

FloatToStrF, 142

функция (*продолжение*)

Floor, 609

Format, 143, 144

FormatDateTime, 123, 143

FormatFloat, 143

frac, 121

FutureValue, 611

GradToRad, 608

hi, 112

High, 119

Hypot, 607

inc, 112

int, 121

InterestPayment, 611

InterestRate, 611

InternalRate, 611

IntPower, 609

IntToHex, 143

IntToStr, 143

Ldexp, 609

Length, 97, 139, 140

ln, 121

lo, 112

Low, 119

LowerCase, 140

MapViewOfFile, 206

Max, 609

MaxIntValue, 610

MaxValue, 610

Mean, 610

MessageBeep, 412

MessageBox, 463

Min, 610

MinIntValue, 610

MinValue, 610

NetPercentValue, 611

Norm, 610

Now, 123

NumberOfPeriods, 6121

ord, 97, 111

Payment, 611

PeriodPayment, 612

PersentValue, 612

pi, 121

PlaySound, 412

Poly, 609

PopnSrdDev, 610

PopnVariance, 610

Pos, 141

Power, 609

Pred, 111

RadToCycle, 608

RadToDeg, 608

RadToGrad, 608

RandG, 610

функция (*продолжение*)

Random, 82
 random, 113, 121
 SeekEof, 200
 SeekEoln, 200
 SetLength, 127
 SetWindowsPos, 440
 sin, 121
 SinCos, 607
 SizeOf, 150
 SLNDepreciation, 612
 sqr, 113, 121
 sqrt, 121
 StdDev, 610
 StrToCurr, 141
 StrToDate, 141
 StrToFloat, 141
 StrToInt, 77, 141
 StrToIntDef, 142
 StrToIntRange, 142
 StrToTime, 142
 Succ, 111
 Sum, 610
 SumInt, 610
 SumOfSquares, 610
 swap, 113
 SYDDepreciation, 612
 Tan, 607
 TanH, 608
 Time, 123
 TimeToStr, 123, 143
 TotalVariance, 610
 Trim, 77
 UnMapViewOfFile, 207
 UpCase, 116
 UpperCase, 141
 VarArrayCreate, 189
 VarArrayDimCount, 189
 VarArrayHighBound, 189
 VarArrayLock, 189
 VarArrayLowBound, 189
 VarArrayOf, 189
 VarArrayRedim, 189

функция (*продолжение*)

VarArrayRef, 189
 VarArrayUnlock, 189
 VarAsType, 188
 VarFromDateTime, 188
 Variance, 610
 VarIsEmpty, 188
 VarIsNull, 188
 VarToDateTime, 188
 VarToStr, 188
 VarType, 188
 WinExec, 489

Х

хранилище
 изображений, 356
 объектов, 55, 59, 569

Ц

цифра, 103

Ч

число
 вещественное, 105
 целое, 105
 шестнадцатеричное, 105

Ш

шаблон
 ввода, 320
 связь с проектом, 295
 создание, 293
 шестнадцатеричная цифра, 103
 шрифт, 243

Э

экземпляр класса, 45
 элемент, 264

Я

язык
 Delphi, 45
 Object Pascal, 34
 ассемблера, 503

Фаронов Валерий Васильевич

**Delphi. Программирование на языке высокого уровня:
Учебник для вузов**

Заведующий редакцией

А. Сандрыкин

Ведущий редактор

Ю. Суркис

Литературный редактор

А. Жданов

Художник

С. Маликова

Корректоры

А. Моносов, И. Смирнова

Верстка

Л. Харитонов

Подписано к печати 01.07.09. Формат 70×100/16. Усл. п. л. 51,6.

Доп. тираж 3000. Заказ №17044

ООО «Лидер», 194044, Санкт-Петербург, Большой Сампсониевский пр., д. 29А.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Отпечатано с фотоформ в ОАО «Печатный двор» им. А. М. Горького

197110, Санкт-Петербург, Чкаловский пр., д. 15.

Delphi. Программирование на языке высокого уровня



Валерий Васильевич Фаронов 25 лет проработал преподавателем на кафедре «Системы автоматического управления» в МВТУ им. Н. Э. Баумана. В настоящее время — консультант и разработчик систем управления базами данных, преподаватель на курсах по Delphi, технический писатель. Свою первую книгу «Программирование на персональных ЭВМ в среде Турбо Паскаль» написал еще в 1989 году. С тех пор вышло более 20 книг, посвященных главным образом программированию в Turbo Pascal и Delphi.

Delphi — не только прекрасный инструмент разработки программного обеспечения, но и язык программирования, позволяющий писать как небольшие программы и утилиты для персонального использования, так и крупные корпоративные системы, работающие с базами данных на различных платформах, распределенные приложения и web-сервисы. В книге рассматривается среда разработки Delphi: галереи компонентов, хранилища объектов, вспомогательный инструментарий, дается описание языка Delphi (в шестой и более ранних версиях он назывался Object Pascal), включая компоненты программ, динамически подключаемые библиотеки, интерфейсы, технологию COM и систему ModelMaker.

Базовый курс для студентов высших учебных заведений, обучающихся по направлению «Информатика и вычислительная техника»



Заказ книг:

197198, Санкт-Петербург, а/я 619
тел.: (812) 703-73-74, postbook@piter.com

61093, Харьков-93, а/я 9130
тел.: (057) 758-41-45, 751-10-02, piter@kharkov.piter.com

www.piter.com — вся информация о книгах и веб-магазин

ISBN 978-5-8046-0008-3



9 785804 600083